AFRL-AFOSR-UK-TR-2011-0047

Plan Representations for Distributed Planning and Execution

Gerhard J Wickler

University of Edinburgh
AIAI, School of Informatics
Appleton Tower, Crichton Street
Edinburgh, Scotland, United Kingdom EH8 9LE

EOARD GRANT 09-3090

August 2011

Final Report for 19 July 2009 to 19 July 2011

Air Force Research Laboratory
Air Force Office of Scientific Research
European Office of Aerospace Research and Development
Unit 4515 Box 14, APO AE 09421

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. REPORT DATE ***(DD-MM-YYYY)*** 15-08-2011	**2. REPORT TYPE** Final Report	**3. DATES COVERED** *(From – To)* 19 July 2009 – 19 July 2011

4. TITLE AND SUBTITLE **Plan Representations for Distributed Planning and Execution**	**5a. CONTRACT NUMBER** FA8655-09-1-3090
	5b. GRANT NUMBER Grant 09-3090
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Dr. Gerhard J. Wickler	**5d. PROJECT NUMBER**
	5d. TASK NUMBER
	5e. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Edinburgh Appleton Tower, Crichton Street Edinburgh, Scotland , United Kingdom EH8 9LE	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) EOARD Unit 4515 BOX 14 APO AE 09421	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/AFOSR/RSW (EOARD)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) **AFRL-AFOSR-UK-TR-2011-0047**

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report results from a contract tasking University of Edinburgh as follows: The final project report describes work completed towards a new framework for distributed multi-agent planning and plan execution. The first step focuses on the plan representation that needs to be sufficiently rich to allow the sharing of plans between humans, software systems, and robotic entities. The basis for this representation is the Core Plan Representation developed in the mid 90s. This representation and its roots in Artificial Planning research are examined in section 3. The focus of this representation is the concept of a plan. Plans are executed by agents, and one of the dominant models of agency in agent research describes three mental attitudes of an agent: beliefs, desires and intentions. In section 4 we shall examine these concepts attempt to merge them with the plan representation. The combined ontology will then be further refined in section 5, introducing concepts that will be required for a sharable plan representation intended for distributed multi-agent planning and plan execution. The report then goes on to describe the software support for this representation, which has been developed as an extension to MediaWiki. The next part of this report defines a number of features that can be used to characterize planning domains, namely domain types, relation fluency, inconsistent effects and reversible actions. These features can be used to provide additional information about the operators defined in a strips-like planning domain. Furthermore, the values of these features may be extracted automatically; efficient algorithms for this are described in this report. Alternatively, where these values are specified explicitly by the domain author, the extracted values can be used to validate the consistency of the domain, thus supporting the knowledge engineering process. This approach has been evaluated using a number of planning domains, mostly drawn from the international planning competition. The results show that the features provide useful information, and can highlight problems with the manual formalization of planning domains. In the final part of this report the focus will be on communication about activity and plans. After an analysis of existing agent communication standards and interaction protocols defined in these approaches we will define a small number of more complex protocols for the support of distributed plan execution. This will be concluded with a discussion on execution failure and a fundamental problem that underlies all existing approaches.

15. SUBJECT TERMS

EOARD, Planning, Distributed Artificial Intelligence, C4ISR

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18, NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT UNCLAS	**b. ABSTRACT** UNCLAS	**c. THIS PAGE** UNCLAS	SAR	50	JAMES LAWTON Ph. D.
					19b. TELEPHONE NUMBER *(Include area code)* +44 (0)1895 616187

Plan Representations for Distributed Planning and Execution[*] Final Report

Gerhard Wickler
AIAI, University of Edinburgh
Edinburgh, Scotland

August 4, 2011

Summary

This report describes work completed towards a new framework for distributed multi-agent planning and plan execution. The first step focusses on the plan representation that needs to be sufficiently rich to allow the sharing of plans between humans, software systems, and robotic entities. The basis for this representation is the Core Plan Representation developed in the mid 90s. This representation and its roots in Artificial Planning research are examined in section 3. The fucus of this representation is the concept of a *plan*. Plans are executed by agents, and one of the dominant models of agency in agent research describes three mental attitudes of an agent: *beliefs*, *desires* and *intentions*. In section 4 we shall examine these concepts attempt to merge them with the plan representation. The combined ontology will then be further refined in section 5, introducing concepts that will be required for a sharable plan representation intended for distributed multi-agent planning and plan execution. The report then goes on to describe the software support for this representation, which has been developed as an extension to MediaWiki.

The next part of this report defines a number of features that can be used to characterize planning domains, namely *domain types*, *relation fluency*, *inconsistent effects* and *reversible actions*. These features can be used to provide additional information about the operators defined in a STRIPS-like planning domain. Furthermore, the values of these features may be extracted automatically; efficient algorithms for this are described in this report. Alternatively, where these values are specified explicitly by the domain author, the extracted values can be used to validate the consistency of the domain, thus supporting the knowledge engineering process. This approach has been evaluated using a number of planning domains, mostly drawn from the international planning competition. The results show that the features provide useful information, and can highlight problems with the manual formalization of planning domains.

In the final part of this report the focus will be on communication about activity and plans. After an analysis of existing agent communication standards and interaction protocols defined in these approaches we will define a small number of more complex protocols for the support of distributed plan execution. This will be concluded with a discussion on execution failure and a fundamental problem that underlies all existing approaches.

[*]Effort sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-09-1-3090. The U.S. Government is authorized to reproduce and distribute reprints for Government purpose notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

Contents

List of Figures

Acknowledgement of Sponsorship: An acknowledgement of Government support appeared in any publication of any material based on or developed under this project, in the following terms: "Effort sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-09-1-3090. The U.S. Government is authorized to reproduce and distribute reprints for Government purpose notwithstanding any copyright notation thereon."

Disclosure of inventions: I (Dr. Gerhard Wickler) certify that there were no subject inventions to declare during the performance of this grant.

1 The Representation

Planning is commonly associated with intelligent behavior in agents [Russell and Norvig, 2003]. The activity of planning can be defined as an *explicit deliberation process that chooses and organizes actions by anticipating their outcomes* and which *aims at achieving some pre-stated objectives* [Ghallab *et al.*, 2004]. Planning in Artificial Intelligence (AI) is the computational study of this deliberation process.

The outcome of the planning process and the solution to a planning problem is a plan, i.e. an organized collection of actions. Most research in AI planning has focussed on algorithms for finding plans, which has proved to be a computationally very hard problem. Assuming that an agent wants to achieve the objective that underlies the planning problem, it will also need to execute the plan. Assuming further that more than one agent is involved in the planning and execution, the plan will need to be communicated between the different agents, calling for a rich and sharable representation of a plan.

2 The Representation: Methods, Assumptions, and Procedures

To this end, the Core Plan Representation (CPR) [Pease and Carrico, 1996] was developed. The CPR establishes a conceptual framework for the rich representation of plans that are to be communicated between agents. The concepts provided are generic in the sense that they will be important for most uses of a plan, be it for execution or any other purpose. Agents using these concepts in the representation of their plans avoid ambiguity as the CPR is a standard that defines these concepts. However, the CPR concepts are focussed on **plans** and the fact that these plans may be executed by agents is almost incidental.

In this research we will look at the concept of a plan as it is defined in the CPR, but we will shift the emphasis to the agents that work with this plan. To achieve this aim, we will attempt to merge the main concepts of the CPR with the core mental attitudes established in research into intelligent agents: Beliefs, Desires and Intentions (BDI). Furthermore, we will refine a number of the concepts involved. This will allow for an even richer representation for communicating plans between agents.

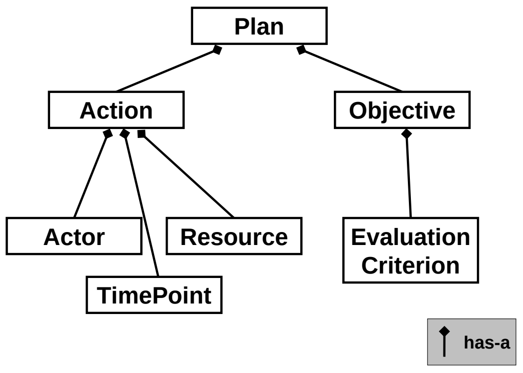


Figure 1: Main concepts of the CPR

The ultimate aim here is to support the distributed execution of a plan (and its sub-plans) by a group of agents in a dynamically changing environment. However, this aspect will be discussed in future reports.

3 The Core Plan Representation

The prime motivation for the development of the CPR was to address the plan interchange requirements of several military planning systems. Just like there are a number of different planning problems and approaches in AI, there are different representations that come with the different systems that implement the various approaches, perhaps even more. Furthermore, systems that process plans, e.g. workflow systems, control systems, etc., will need to exchange information with the AI plan-generating systems. And we must not forget the human in the loop: people need to understand plans in order to execute them in line with the intent that underlies them.

The result of the CPR effort is a small ontology that defines a few core concepts that are expected to be shared between most systems that deal with plans. The way these concepts are defined relies on ontological principles: each concept is defined by the relations that must hold with other concepts and some internal structure. Furthermore, there is human-readable text that explains the concepts, but this can usually not be processed by any planning system.

The main concepts that form the CPR are shown in figure 1. The most important relation that holds between these concepts is the "has a" relation that indicates that an instance of one concept has a component that is an instance of the other concept. Alternatively, the second concept forms a building block for the former. From an ontological point of view this is slightly unusual as, in most ontologies, the "is a" relation is the most prominent relation.

3.1 Plans

The central concept in the CPR is of course the plan, the thing the CPR is meant to represent. A plan is also the output of a classic AI planning system, where a plan is an organized collection of actions. In the CPR, the definition of a plan includes more:

- **name**: a unique reference that can be used to refer to a specific plan;
- **sub-plans**: a set of finer grained plans which correspond to a hierarchical decomposition of the overall plan;
- **objectives**: a set of objectives that are accomplished by the plan;
- **actions**: a set of actions which must be performed as part of this plan;
- **alternatives**: that names of other plans (that were considered);
- **evaluation**: the merits of this plan (compared to its alternatives); and
- **issues**: metadata about the status of plan creation.

Explicit naming of plans is necessary for communication purposes. The inclusion of sub-plans indicates that the approach is based on a classic HTN planning paradigm [Tate, 1977], where a planning problem is given as a task (an activity) to be accomplished. However, the nature of objectives is not well defined in the CPR: objectives could be tasks as in HTN planning or they could be state-related goals as in STRIPS planning [Fikes and Nilsson, 1971]. We shall assume that objectives are meant to include both. Actions will be considered in detail below. The next two components, alternatives and merits, are usually not considered in classic planning, but are of great practical relevance during the planning process. Finally, issues in CPR are a generalization of flaws in AI planning.

3.2 Actions

Actions are the main components in a plan. They correspond to activities an agent can do. While plans are considered too complex to be directly executable by and agent and hence require being broken down, actions can be considered primitive in that an agent does not need to be told as part of the plan how the action is to be done, i.e. further decomposition is optional. In the CPR, an action consists of:

- **name**: a unique reference that can be used to refer to a specific action in a plan;
- **sub-actions**: a set of finer grained actions which correspond to a hierarchical decomposition of the action (e.g. if there are multiple actors) ;
- **actors**: a set of agents that will execute the described action;
- **resources**: a set of (reusable and consumable) resources that will be used during the execution of this action;
- **plan objects**: a set of objects that function as parameters to this action; and
- **begin** and **end**: two time points representing the beginning and the end of this action.

Again, explicit naming of actions is useful for communication. The decomposition of actions into sub-actions is odd from a planning perspective: what is the difference between a plan that requires decomposition and an action that requires decomposition? One difference is that actions are associated with actors, the agents that execute the action. Furthermore, actions are associated with resources used by the action, giving an action a more concrete feel than a plan, and making them the subject of scheduling algorithms rather than planning in AI terminology. The same is true for beginning and end time points, which are usually assigned by schedulers.

3.3 Actors

While plans and actions are concepts firmly rooted in classic AI planning, the concept of an actor has not received much attention in the planning literature. However, there is another area of AI, multi-agent systems [Wooldridge and Jennings, 1995; Wooldridge, 1999], in which the concept of an agent is central. More on this below. In the CPR, an actor is a relatively simple concept:

- **name**: a unique reference that can be used to refer to a specific actor;
- **objectives**: a set of of objectives this actor shares with the plan; and
- **sub-actors**: a set of actors if this is an aggregate actor.

The objectives here are a subset of the objectives that are associated with a plan to which an action assigned to this actor belongs. This allows for the limited representation of rationale in the plan, which can be most useful for deciding how exactly to implement an action or during execution failure. The representation of sub-actors allows for the representation of hierarchical organizations. Whether an actor has authority over its sub-actors is not defined in the CPR.

3.4 Resources and Time Points

Resources are not defined formally in the CPR. They are given a name that can be used to refer to them, and they are used by actions, but other than that, it is up to the planner to use them in a meaningful way. Again, AI planning does not deal much with resources as scheduling algorithms tend to be much more efficient in managing them.

The same applies to time points in CPR, except that they are not even given a name. This may be due to the fact that time has a common sense meaning and is exactly that which is used here.

3.5 Objectives and Evaluation Criteria

Objectives come in two forms in AI planning, as mentioned above, either as tasks to be performed (activities) or goals to be achieved (world state conditions). Either way, they are part of the planning problem, but usually not part of the plan that is a solution to a planning problem.

In the CPR an objective contains the following attributes: type, value, actions, evaluation criteria, and sub-objectives. The actions are those that contribute to this objective. They may also contribute to other objectives, and actors that execute these actions should have this objective as one of their objectives to be consistent. Associated with an objective is a set of evaluation criteria.

Like resources and time points, evaluation criteria are not defined in terms of attributes within the CPR.

4 Beliefs, Desires and Intentions

A BDI agent is distinguished by the organization of its knowledge, which governs its behavior, into three distinct knowledge structures based on the mental state modalities: *beliefs*, *desires* and *intentions* [Rao and Georgeff, 1991]. Beliefs represent the agent's current knowledge about itself and its environment, desires represent its longer term goals and objectives of its behavior, and intentions represent the agent's local decisions about the actions it intends to perform.

4.1 Semantics

The three core concepts that make up the BDI model are not defined in the style of an ontology, but rather through an operational semantics. A BDI agent executes its behavior by manipulating its internal knowledge and data according to the standard BDI inference loop:

```
function BDI-inference-loop($Bel, Des, Int$)
    while true do
        $p \leftarrow$ getNextPercept()
        $Bel \leftarrow$ beliefRevision($Bel, p$)
        $Int \leftarrow$ generateOptions($Bel, Des$)
        $Int \leftarrow$ filterIntentions($Bel, Des, Int$)
        $plan \leftarrow$ generatePlan($Bel, Int$)
        execute($plan$)
```

A BDI agent first processes the next external percept (such as an incoming message). As a result the agent revises its beliefs. Then, based on the new beliefs, it generates several different intentional options out of which one is adopted by means of the *filterIntention* function. Next, the *generatePlan* function elaborates a plan for the adopted intentions that is then executed. In applications of the BDI model this planning process is often based on pattern-matching against a library of predefined plans.

The BDI model is one of an agent that fulfills three key qualities: autonomy, reactivity and intentionality [Wooldridge and Jennings, 1995]. The BDI model does not explicitly support the social properties of an agent: the ability to communicate, cooperate and reason about other agents in a multi-agent community.

4.2 Task-Centric BDI

Implementations of the BDI model are often very simple in that they are based on a propositional representation: beliefs are sets of propositional symbols that hold in a given world state; desires and intentions are also propositions, namely the conditions the agent desires or intends to make true in a future state of the world. Defined like this, the BDI model can be reasoned about with classic theorem provers quite easily. However, a more complex implementation, e.g. using first-order atoms as beliefs means current theorem provers are insufficient to work as interpreters for an agent specified at the BDI level.

For this reason we have adopted a task-centric view of an agent, in which beliefs are essentially sets of first-order atoms, and desires and intentions are described by tasks to be accomplished and plans to be executed respectively [Wickler *et al.*, 2007]. In this view, an HTN planner can be used as an agent interpreter.

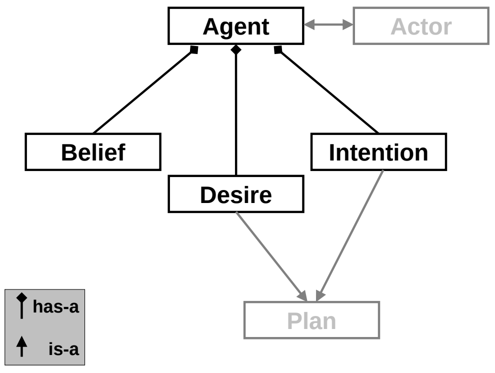


Figure 2: Main concepts of the BDI model

5 Combining CPR and BDI

In this section we will present a combined view of the main concepts from the CPR and the BDI agent model. We will then continue to extend the model by refining the different concepts in order to enrich the representation. The aim is to come up with a model that allows semantically rich plans to be shared.

5.1 Ontological View

Despite the name (BDI), the central concept of this model is the agent. An agent is defined by its mental attitudes just like a plan in the CPR is defined by its attributes. Thus, an agent is defined by a set of beliefs, a set of desires, and a set of intentions, as shown in figure 2.

The concept of a BDI agent represents essentially the same concept as a CPR actor. In both cases, these are the entities that execute actions and deliberately manipulate the world according to some mental model. There are some differences though: In the BDI model the mental attitudes are the focus, whereas the CPR only mentions the objectives of an agent, and only those relevant to the plan in which the actor occurs are considered. The BDI model on the other hand ignores the internal struc-

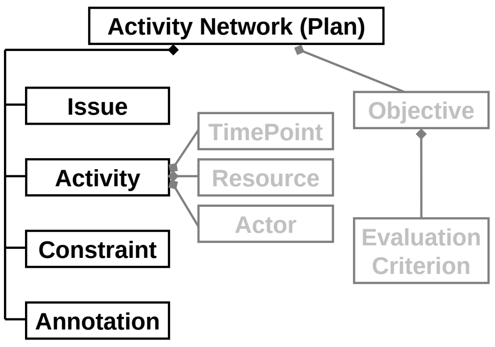


Figure 3: Plans as <I-N-C-A> objects

ture of an agent that may be an organization with sub-actors; it only considers individual agents.

Objectives can be tasks to be accomplished or goals to be achieved. They correspond to desires in the task-centric or the classic BDI model respectively. In HTN planning there is no real distinction between a task which can be given as input to a planner and a plan that is the corresponding output; both are plans. The difference is that a task usually requires refinement before it becomes an executable plan, and that is exactly what refinement planners do [Kambhampati *et al.*, 1995].

5.2 Refining the Concepts

The nine concepts introduced so far can be used as the core of a sharable plan representation. In the remainder of this section we will refine and extend these concepts by extending the ontology. This will include relations that must hold between the various concepts, resulting in a more tightly defined semantics.

5.2.1 Plans as Activity Networks

The first refinement to be introduced here is based on the <I-N-C-A> ontology [Tate, 2003] which describes a plan consisting of four main components: issues, nodes, constraints and annotations. Despite different terminology used, the <I-N-C-A> model is really an extension of the CPR model, where both, an <I-N-C-A>object and a plan are in fact a specification of an activity network, and this is the term we have used in figure 3.

Given that the <I-N-C-A> model is meant for describing any type of synthesized object, not just a plan, its terminology is also more general, referring to the synthesized components as *nodes*. We can be more specific here: in a plan the synthesized components are activities. An activity corresponds directly to an action in the CPR. However, the term *action* has been heavily overloaded in the planning literature, making it more convenient to adopt the term activity here. Activities will be discussed in detail below.

A second component already included in the CPR is the constraint. However, we omitted this concept here as it was included in the CPR ontology but not related to any of the other concepts. <I-N-C-A> has constraints as main components of a plan. In fact, the objective of a plan can be itself a constraint, one that must be included in a plan. This is coherent with the view that a plan is a specification of behavior, and the objective is a special constraint, namely that any refinement of the specification must include the achievement of the objective.

Issues have no equivalent concept in CPR or BDI. One interpretation of an issue is a meta-level activity, i.e. something that needs to be done at the planning level. For example, flaws that need to be resolved can be seen as issues that exist within a plan. A more general view in line with the general synthesis problems for which <I-N-C-A> is suitable, an issue is a question [Conklin, 2005] that needs to be answered about the plan.

Finally, annotations can be used to capture any additional information about the plan or elements of it, e.g. rationale [Wickler *et al.*, 2006].

5.2.2 Actions and Activities

Activities are an extension of actions in the CPR and the new components are shown in figure 4. As in the CPR, we must have an actor (agent) associated with an activity. This may be the agent executing the activity, or at least being responsible for it. An activity may rely on a given set of resources, and an activity has at least two time points associated with it, the beginning and the end. Further time point may be specified depending on the complexity of the representation.

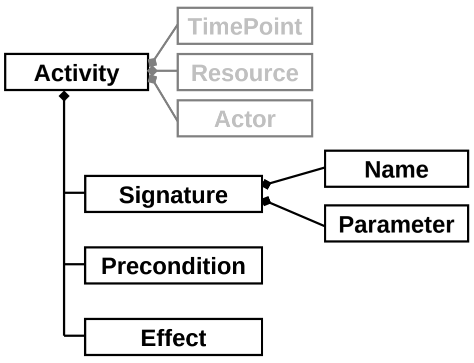


Figure 4: Refined activities

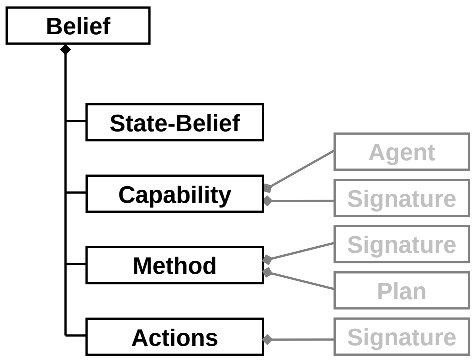


Figure 5: Types of belief of an agent

From the more traditional view of planning we have also taken the signature, preconditions and effects as elements of an activity. The signature is simply a refinement of the name attribute that is defined in the CPR. The signature itself consists of a name that denotes the activity type, corresponding to a STRIPS operator name, and a set of (typed) parameters representing the objects manipulated by this activity. In the CPR these objects were directly associated with the activity as plan objects.

Preconditions and effects are again taken from a STRIPS planning view and there did not appear to be a corresponding field in the action of the CPR. Both are world state conditions, usually represented as atoms of first-order logic. More complex representations are possible here. If there is an agreed ontology of activity types that is shared between all agents, then there is no need to explicitly include preconditions and effects in the plan as they are indirectly available through the ontology.

The reason why they are important is that preconditions need to be verified at execution time: when an agent is about to start an activity it should check that all the preconditions hold to ensure that the context for executing the activity in the current plan is indeed given. Similarly, when the activity is completed, the agent should check that all the required effects have been achieved.

5.2.3 Types of Beliefs

Beliefs are the statements an agent believes to hold in the world at given point in time. For the a plan to be shared it is important for the agents involved to share certain beliefs, which is why these should be included in a rich plan representation. An overview of the types of beliefs we have identified is given in figure 5.

Firstly, there are beliefs about the world state. This is factual knowledge about relations that hold between objects at specific points in time. These can be in the past, present, or future. The knowledge is static in the sense that it represents time slices of world states and does not include relations that go from one time slice to another.

Second, there is knowledge about capabilities of different agent. In a collaborative environment agents need to know what others are capable of doing. Thus, each capability is associated with an agent. The capability itself is given as the signature of the activity the agent is capable of performing. The signature effectively is a pointer to a more elaborate description in the shared ontology. It is not expected that the capability is always available, or in all circumstances. To use the capability of another agent in a plan, some negotiation with that agent needs to take place, either at planning time, or at execution time which can lead to failure of course.

Next, there are methods known to an agent. This is procedural knowledge that is necessary during the (HTN) planning process. Technically, there are sometimes known as refinements as they describe how a complex activity can be refined into something that is closer to an executable plan, e.g. by breaking the activity down into sub-activities that are organized in some way. Thus, the result of the refinement is itself an <I-N-C-A> object, i.e. a plan. In the procedural knowledge of the agent this is associated with the activity that can be refined by this method.

Finally, an agent needs to know what activities it can execute directly, i.e. without further planning. Each of these is described by the signature of the activity.

Note that this representation does not include knowledge about knowledge of other agents. While this is sometimes necessary for reasoning, it is also a complex problem that we con not address adequately yet.

5.2.4 Desires and Intentions

As mentioned above, desires and intentions are both represented as activity networks (plans) in this representation. Both can be seen as an agenda though which agent behavior is guided. The difference lies in the fact that the agent holding the desires and intentions is committed to only the intentions. The commitment may be to itself, or it may be to one or more other agents. In the latter case an activity that is a commitment cannot simply be dropped, but there must be a way of negotiating this with the other agents the commitment is to. There are no commitments with respect to desires.

Note that a special type of activity is *planning*. Thus, and agent may have an intention to plan for one of its desires (at a future point in time). This can then result in the agent adopting that plan as an intention, scheduled to be executed. If this new plan contains another *planning* activity, this approach can be used to control a robotic agent.

5.2.5 Execution Status

The final concept to be described here is not mentioned in either the CPR or the BDI model, but it is clearly relevant for plan execution: the execution status of a plan. As this a rather simple component at this stage, no figure is necessary to describe it. The execution status of a plan is simply defined as the completed actions, the actions in progress, and the actions still to do. Each of these actions must be associated with an agent in the plan.

If the plan can be executed without problems, this simple structure will suffice to describe the execution status os a plan. However, plan plans tend to go wrong, especially in military domains where adversaries have an active interest in disrupting plans. This means plan repair and re-planning may need to occur as part of the plan execution. The plan execution status must capture this. Thus, further research is required to establish concepts and structures that can capture this process and the (dynamic) plan adequately.

6 The Implementation

The sharing of information online has progressed significantly with the advent of Web 2.0 technology, where the content of websites can be dynamically updated by the users of that site. One of the technologies that implements this approach are wikis, of which MediaWiki [Barrett, 2008] is one of the most popular and robust. Information in a wiki is usually more or less free form, which has the advantage that anything can be written down and shared. The drawback is that unstructured information allows for only limited automated processing.

7 Implementation: Methods, Assumptions, and Procedures

To address this issue, MediaWiki has an extension mechanism that allows for arbitrary XML tags to be defined, and this is the approach we have chosen to develop a semi-formal software tool that supports the development of rich and sharable planning and plan information. The concepts identified in the first report are implemented as XML elements that can be used in an otherwise free-form document. This means people can develop plans and planning knowledge even if they are not experts in the formal aspects of the representation. Other people could then mark up the content using the formal concepts provided by the CPR/BDI representations. MediaWiki provides the mechanisms for collaborative article development and sharing.

This report describes the implementation of the different elements that come with the extension to MediaWiki. It can be seen as a reference manual that illustrates the different concepts using a relatively simple domain that is used in the literature on AI planning.

8 Representing Procedural Knowledge in a Wiki

In this section we will describe the formal syntax of a mark-up language that has been implemented as part of the extension of MediaWiki [Barrett, 2008]. This language defines a number of XML tags that can be used give wiki text describing procedural knowledge a formal meaning that can then be exploited for formal reasoning about the procedures represented in the wiki articles. The concepts that are represented by the formal language were described by a semi-formal ontology in [Wickler, 2010], and an overview of these concepts is repeated here in figure 6

The implementation of most of these concepts involves three steps. Firstly, the XML tags that define wiki text to have a specific formal meaning must be declared. These tags can then appear in the source of a wiki article. We shall illustrate the use of these tags with various examples. Secondly, the marked up text that represents instances of concepts from the CPR/BDI ontology in figure 6 must be stored in a database. This has been accomplished by adding a set of new tables to the database used by MediaWiki, and the SQL statements used to create these tables will be given as precise definitions of the entities extracted from an article. Finally, the source wiki text has be rendered to make it readable for an average user not familiar with wiki text or the XML tags described here. We shall used a series of sample screen shots to illustrate how the text is currently rendered.

8.1 Fundamentals: Conditions

Before we develop the different concepts from the CPR/BDI ontology, however, we shall look at one fundamental concept that is shared by many of the more complex structures: a logical literal. In first-order logic this represents an atomic relation between several arguments, or a negation of such an atomic relation. The relation itself is represented by a logical symbol, i.e. a name used to refer to this relation. The arguments typically represent objects that exist in the domain of interest, and they too are given names that can be used to refer to them. The following represents two such atoms representing world state conditions:

```
<atom>
  <rel name="belong" />
  <object name="k1" />
  <object name="l1" />
</atom>;
<atom>
  <rel name="belong" />
  <object name="k2" />
  <object name="l2" />
</atom>;
```

This is example is taken from a toy domain commonly used in the AI planning community: the dock worker robots domain [Ghallab *et al.*, 2004]. The `rel` XML element defines a logical relation that must have exactly one attribute, its `name`, which can be an arbitrary string. The arguments to this relation are defined using the `object` element which also assumes one attribute `name`. Relations and objects are grouped together in an `atom` element. The example states that there is a crane (`k1`) which is at location `l1`, and another crane (`k2`) at location `l2`.

Logical literals like this are used in world state information, preconditions and effects of actions, goal descriptions, and a number of other concepts from the CPR/BDI ontology. The are stored in a table of conditions that is crested as follows:

```
create table mw_plan_cond (
  aid int unsigned not null,
  cid int unsigned not null,
  sign bool not null,
  cond varchar(128) not null,
  role int unsigned not null,
  owntype int unsigned not null,
  ownid int unsigned not null,

  primary key (aid, cid)
);
```

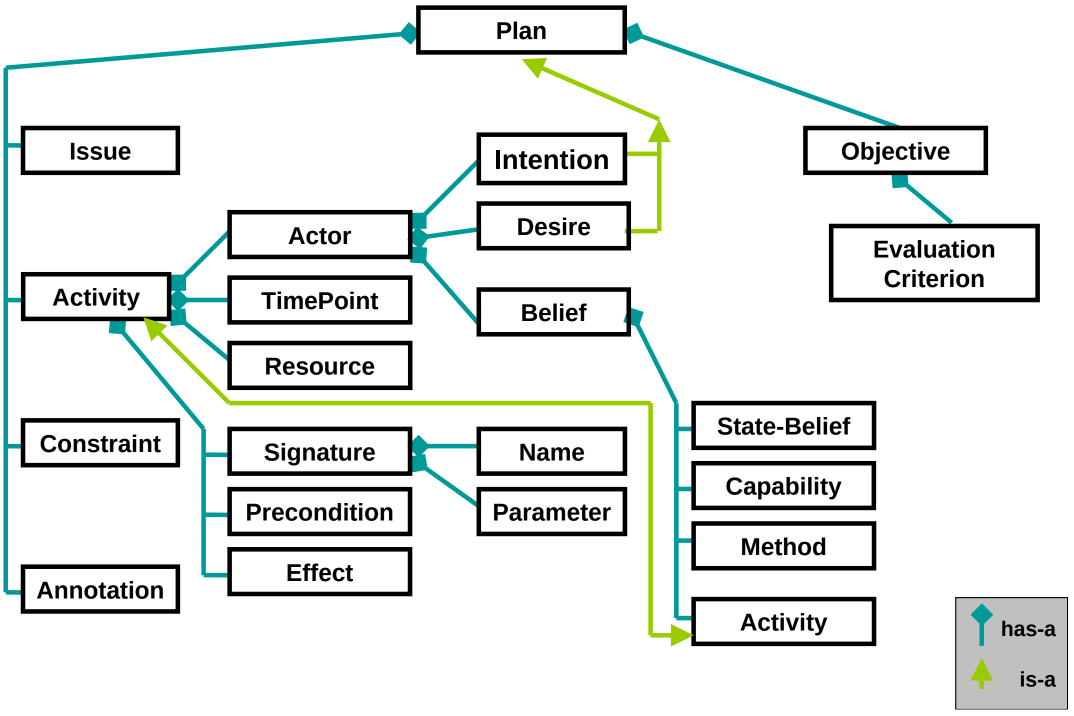


Figure 6: Main concepts of the CPR/BDI Representation

The aid field is defined for every table holding instances of procedural knowledge and simply references the article in which a specific knowledge element (here, a logical condition) has been defined. The cid is the unique index of this condition. The condition itself consists of a sign indicating whether this is a positive or negative atom, and a string representation of the condition in the field cid. The role indicates how this condition is used, e.g. as a precondition or as an effect. Finally, there is information about the container of this condition, namely its type in owntype and the unique reference of the owner in ownid.

Many examples of rendered conditions will follow in the remainder of this report.

8.2 Planning Operators

Operators are one of the oldest structures that were developed for the representation of basic activities. In the literature, they are also referred to as action types, or simply actions. The latter is often used to refer to instances of action types in a plan. An operator consists of a name, an ordered sequence of parameters describing the objects that are manipulated by this action type, and a set of preconditions and effects. The following is an example of wiki text that has been marked up as an operator:

```
<operator name="move">
Move a robot from one location to another.

<parameters>
* <var name="r" type="robot" />: the robot
  which will be moved by this action;
* <var name="from" type="location" />: the
  location from which the robot will move
  away (the origin);
* <var name="to" type="location" />: the
  location to which the robot will move
  (the destination).
</parameters>

<preconditions>
* <atom>
  <rel name="adjacent" />
  <var name="from" />
  <var name="to" />
</atom>: the origin and the destination
  have to be adjacent;
* <atom>
  <rel name="at" />
  <var name="r" />
  <var name="from" />
</atom>: the robot has to be at the origin;
* <atom sign="not">
  <rel name="occupied" />
  <var name="to" />
</atom>: the destination must not be
  occupied.
</preconditions>

<effects>
* <atom>
  <rel name="at" />
  <var name="r" />
  <var name="to" />
</atom>: the robot will be at the
  destination;
* <atom sign="not">
  <rel name="occupied" />
  <var name="from" />
</atom>: the origin will no longer be
  occupied;
* <atom>
  <rel name="occupied" />
  <var name="to" />
</atom>: the destination is now occupied
  (by the robot);
* <atom sign="not">
  <rel name="at" />
  <var name="r" />
  <var name="from" />
  </atom>: the robot is no longer at the
  destination.
</effects>

The above definition does not exclude the
origin and destination being the same
location. While this does not constitute
a problem, it may increase the size of
the search space and hence slow down
the planning process.

</operator>
```

The name of the operator is a required attribute of the operator tag and must be unique. This example also shows an important feature of the representation, that is not used much here in the interest of brevity: the tagged information is only part of the representation. Any information that is not directly part of a formal aspect of the description is simply added to the wiki text and not marked up using the XML tags. This text will be rendered by the wiki as if no extension was present. For typical standard operating procedures, it is expected that this constitutes the majority of the description.

The `parameters` are a sequence of variable declarations. Each variable (tagged `var`) must have a `name` and optionally, a `type`. Again, the example shows that normal text can be mixed into the representation to clarify and elaborate.

The `preconditions` and the `effects` of an operator are conditions of the type described in the previous section. As opposed to the parameters, the order of the preconditions and effects is only stored in the source text, but may be different when the operator is exported directly from the database. All the arguments to the various conditions associated with an operator should only use variables that are parameters to the operator.

The database uses two tables to store the operators (plus the conditions table). The first table holds the parameter objects with their respective types. The table is created as follows:

```
create table mw_plan_param (
  aid int unsigned not null,
  pid int unsigned not null,
  name varchar(16) not null,
  type varchar(16),
  pos int unsigned not null,
  owntype int unsigned not null,
  ownid int unsigned not null,

  primary key (aid, pid)
);
```

The `pid` field is a unique reference to this parameter. The `name` and `type` contains the respective attributes from the XML element. The position of the parameter has to be made explicit in the `pos` field as there is no predefined order in the database.

```
create table mw_plan_operator (
  aid int unsigned not null,
  oid int unsigned not null,
  name varchar(64) not null,
  owntype int unsigned,
  ownid int unsigned,

  primary key (aid, oid)
);
```

Since both, conditions and parameters have an explicit link to their owner, the operator itself has only filed that represents information about the operator directly, namely the `name` of the operator. As opposed to conditions and parameters, and operator need not occur as content of some other element, and hence the owner information may be `null`.

When an article that contains an operator is saved, the relevant information for the tables mentioned above is automatically extracted from the article text and written to the database. Before this is done, however, any CPR/BDI objects that have the current article id are deleted from the database. This ensures that updates are handled correctly.

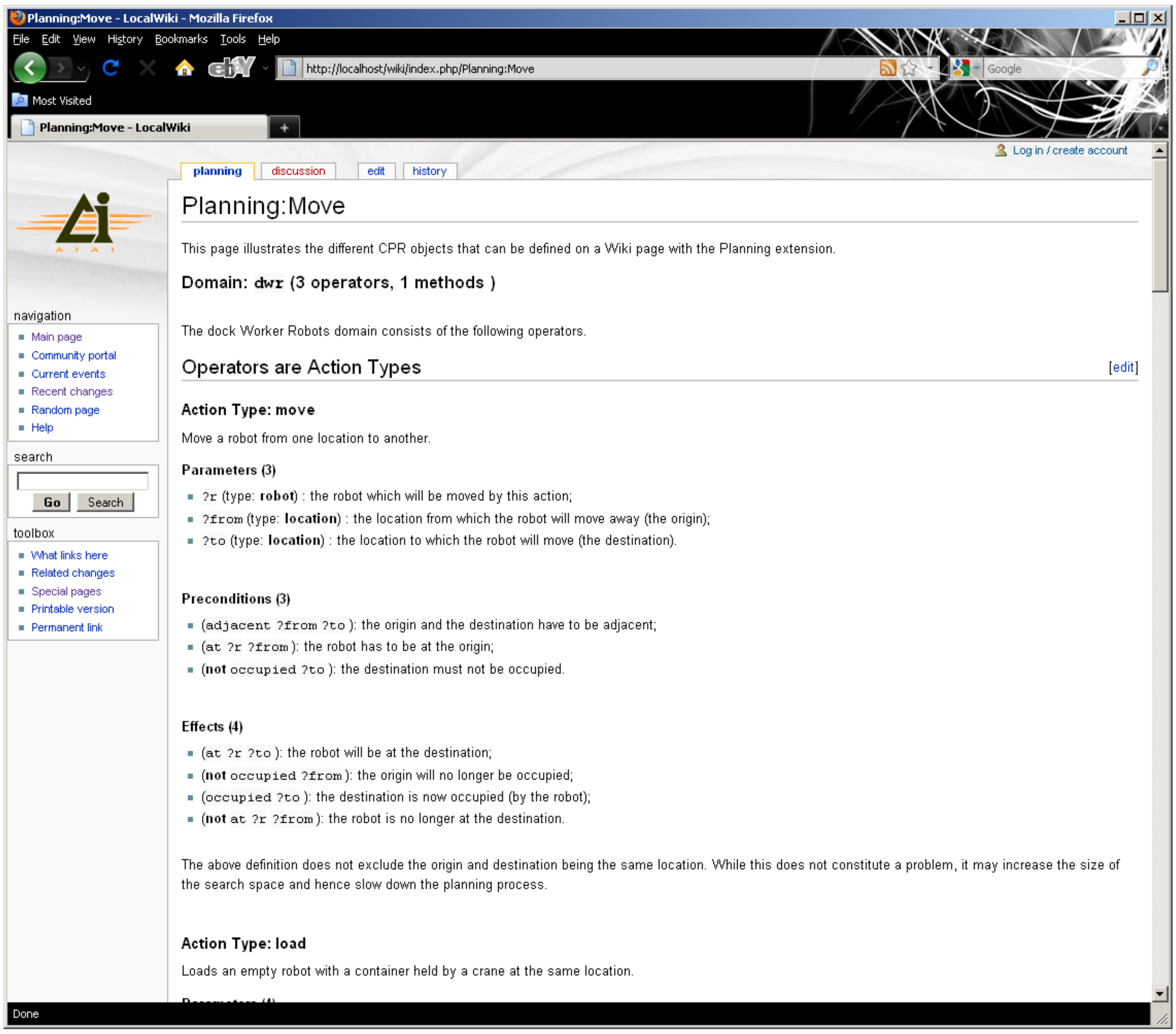
The rendered article containing the move operator is shown in figure 7. The HTML that is generated begins with the heading **Action Type: move**. The parameters, preconditions and effects are listed under the following three sub-headings. The text below the operator is simply a comment and not part of the formal CPR/BDI representation.

8.3 Actions in a Plan

Action types define the way the application of an action changes the world, where an action is an instance of an operator. There can be many actions of the same type in a plan, and in fact, there often are. There can even be multiple actions having the same parameter values in a single plan. The following example shows marked up wiki text that defines a single action:

```
<action type="move"
    ref="CTRL-2011-1"
    agent="controller"
    finish="2012-01-01 00:00:00">

<parameters>
* <object name="r1" />: move robot nr. 1;
* <object name="loc456" />: from the dock
  (the origin);
* <object name="loc789" />: to the yard
```



Planning:Move

This page illustrates the different CPR objects that can be defined on a Wiki page with the Planning extension.

Domain: `dwr` (3 operators, 1 methods)

The dock Worker Robots domain consists of the following operators.

Operators are Action Types [edit]

Action Type: move

Move a robot from one location to another.

Parameters (3)

- `?r` (type: **robot**) : the robot which will be moved by this action;
- `?from` (type: **location**) : the location from which the robot will move away (the origin);
- `?to` (type: **location**) : the location to which the robot will move (the destination).

Preconditions (3)

- (`adjacent ?from ?to`): the origin and the destination have to be adjacent;
- (`at ?r ?from`): the robot has to be at the origin;
- (**not** `occupied ?to`): the destination must not be occupied.

Effects (4)

- (`at ?r ?to`): the robot will be at the destination;
- (**not** `occupied ?from`): the origin will no longer be occupied;
- (`occupied ?to`): the destination is now occupied (by the robot);
- (**not** `at ?r ?from`): the robot is no longer at the destination.

The above definition does not exclude the origin and destination being the same location. While this does not constitute a problem, it may increase the size of the search space and hence slow down the planning process.

Action Type: load

Loads an empty robot with a container held by a crane at the same location.

Figure 7: An Operator rendered in the Wiki

```
  where the containers are (the
  destination).
</parameters>

The robot can now be loaded with a
container.

</action>
```

The action element has a number of required attributes, the first of which is the name of the action type this action instantiates. Since the action type defines the preconditions and effects, there is no need for them in the action. Another attribute is the unique reference of this action, which can be used to distinguish otherwise identical actions in a plan. Also required is a reference to the agent that has to execute the action, which is simply the name of the agent. Finally, the `start` and `finish` attributes are optional, and the example uses only the latter to set a deadline for the action.

The content of an action element is a single list of parameters. Since actions must be fully ground, all the parameters are objects, as described in the conditions above.

The database schema that describes how actions are stored is defined as follows:

```
create table mw_plan_action (
  aid int unsigned not null,
  uref varchar(64) not null,
  descr varchar(255) not null,
  agent varchar(64) not null,
  abeg datetime,
  aend datetime,
  owntype int unsigned,
  ownid int unsigned,

  primary key (aid, uref)
);
```

The `uref` field contains the unique user-defined reference for this action. The description in `descr` contains the action type as well as the parameter values for the action. `agent abeg` and `aend` contain attributes from the action element. Finally, an action may be self contained, or it may belong to another entity, which must be a plan.

The wiki text formally defining an action is rendered by our extension to MediaWiki as shown in the lower part of figure 8.

The time points are defined in a format specific to the database, but this could be formatted in different ways. Again, it should be clear that the rendered text should be accessible to any user, whereas the source text is not. A future version of the MediaWiki extension could also link the symbols that refer to specific objects to web pages that describe these objects, where available.

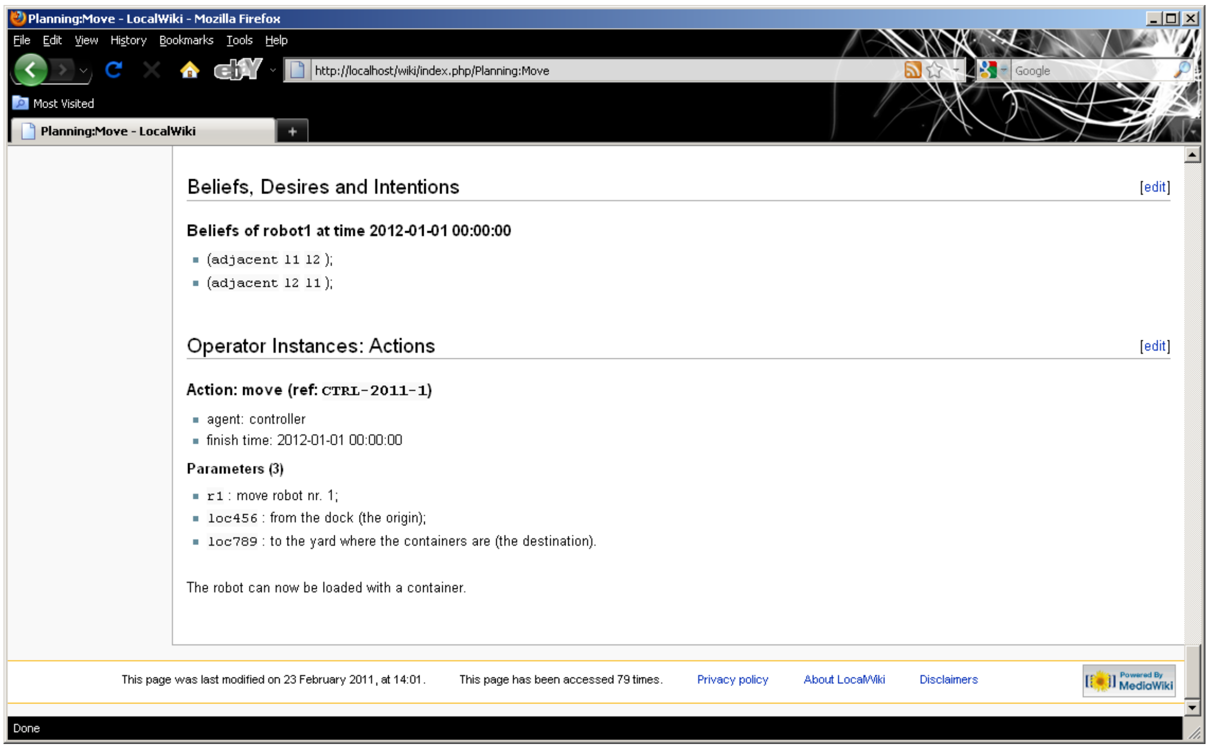
8.4 Hierarchical Refinements

The next concept from the CPR/BDI ontology extends the representation with hierarchical task refinements, an approach to AI planning that has been used by many practical planning systems. This is because the standard operating procedures found in manuals often resemble this style of procedural knowledge. From a technical point of view, the representation is as for operators, namely, a set of XML element tags that can be used to mark up wiki text as a method that breaks down a complex task into finder grained activities. The following is an example of such a refinement:

```
<method name="take-and-put">

<parameters>
* <var name="c" type="container" />: the
  container which is moved from pile to
  pile;
* <var name="k" type="crane" />: the
  crane  which will pick up and put down
  the container;
* <var name="l" type="location" />: the
  location at which all this takes place;
* <var name="po" type="pile" />: the pile
  from which the topmost container is to
  be moved;
* <var name="pd" type="pile" />: the pile
  to which the container is to be moved
* <var name="xo" type="container" />:
  the container on which the container to
  be moved is currently;
* <var name="xd" type="container" />: the
  container onto which the container to
  be moved will be.
</parameters>

<accomplishes>
<activity type="move-top-container">
```


Planning:Move - LocalWiki - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/wiki/index.php/Planning:Move
Most Visited
Planning:Move - LocalWiki

Beliefs, Desires and Intentions

Beliefs of robot1 at time 2012-01-01 00:00:00

- (`adjacent l1 l2`);
- (`adjacent l2 l1`);

Operator Instances: Actions

Action: move (ref: `CTRL-2011-1`)

- agent: controller
- finish time: 2012-01-01 00:00:00

Parameters (3)

- `r1` : move robot nr. 1;
- `loc456` : from the dock (the origin);
- `loc789` : to the yard where the containers are (the destination).

The robot can now be loaded with a container.

This page was last modified on 23 February 2011, at 14:01. This page has been accessed 79 times. Privacy policy About LocalWiki Disclaimers

Figure 8: An Action rendered in the Wiki

```
<parameters>
* <var name="po" />: the pile from which
  the topmost container is to be moved;
* <var name="pd" />: the pile to which
  the container is to be moved.
</parameters>
</activity>
</accomplishes>

<preconditions>
* <atom>
  <rel name="top" />
  <var name="c" />
  <var name="po" />
</atom>: the container must be at the
  top of the pile from which it is to be
  moved;
* <atom>
  <rel name="on" />
  <var name="c" />
  <var name="xo" />
</atom> (used to bind xo);
* <atom>
  <rel name="attached" />
  <var name="po" />
  <var name="l" />
</atom>;
* <atom>
  <rel name="attached" />
  <var name="pd" />
  <var name="l" />
</atom>;
* <atom>
  <rel name="belong" />
  <var name="k" />
  <var name="l" />
</atom>: both piles and the crane must
be at the same location (used to bind l);
* <atom>
  <rel name="top" />
  <var name="xd" />
  <var name="pd" />
  </atom>: (used to bind xd).
</preconditions>

Note that there is no precondition that
requires the crane not to hold a container
already.

<steps>

The first step is to pick up the container
with the crane. As a result the crane will
be holding the container.
<activity type="take" ref="take">
<parameters>
  <var name="k" />,
  <var name="l" />,
  <var name="c" />,
  <var name="xo" />,
  <var name="po" />.
</parameters>
</activity>

The second step is to put down the
container  on the destination pile. As a
result the crane will be empty.
<activity type="put" ref="put">
<parameters>
  <var name="k" />,
  <var name="l" />,
  <var name="c" />,
  <var name="xd" />,
  <var name="pd" />.
</parameters>
</activity>

</steps>

<constraints>
* <atom>
  <rel name="before" />
  <object name="take" />
  <object name="put" />
</atom>: the ordering of the two steps.
</constraints>

</method>
```

The description of a method or refinement is enclosed in a `method` element, which has one attribute, the `name` of the method. The first element inside a method are the parameters, which are exactly the same as for operator descriptions, a sequence of variable declarations.

The next element is used to describe what exactly this method accomplishes. This will be a task marked up with the `activity` element, which, at this point, has only one attribute, namely the `name` of the activity. In the future, this could be used as a reference into an ontology of activities. An activity contains a list of parameters, which should be variables declared for the method.

The preconditions that follow look exactly like the one for an operator.

Instead of effects, however, a method has a set of steps that must be executed in order to accomplish the task for this method. The `steps` are again activities that consist of an `activity` element that contains `parameters`. The difference is that the `activity` elements here also contain a `ref` attribute that can be used to define a unique reference name for this activity in this method.

The final element in a method are the constraints. The example above shows just one constraint that asserts an ordering between the two activities that constitute the steps of this method. The exact types of constraints that are permitted depends on what a reasoning engine, e.g. a planner, can handle. As the wiki simply stores these constraints and does not reason over them, any constraint can be added to the marked up wiki text. And, of course, any information that appears important to the knowledge engineer but does not fit into the tag structure can be added as plain text and will not be lost.

The representation of the methods in the database requires two additional tables that are defined below.

```
create table mw_plan_activity (
  aid int unsigned not null,
  rid int unsigned not null,
  uref varchar(64) not null,
  descr varchar(255) not null,
  owntype int unsigned not null,
  ownid int unsigned not null,

  primary key (aid, rid)
);
```

The first table stores the activities in a method. Mostly, this consists of the user defined references for steps within the method body, and a textual description containing the activity name and its arguments. The owner of an activity must be a method, at present.

```
create table mw_plan_method (
  aid int unsigned not null,
  mid int unsigned not null,
  name varchar(64) not null,
  task varchar(255) not null,
  owntype int unsigned,
  ownid int unsigned,

  primary key (aid, mid)
);
```

Having seen all the elements and structure in the sample method above, it is perhaps surprising to see the simplicity of the table that contains the methods. It has two main fields, the name of the method and the activity that is the task accomplished by the method. All other elements that link into a method specify the method as their owner in the respective tables.

The rendered method is shown in figure 9.

8.5 Classical Planning Domains

Planning domains can be defined in the MediaWiki extension, but consist simply of a named set of planning operators and methods. As this simply adds one XML element that surrounds the other elements described above, there is no need to add a lengthy example here.

Domains are stored in their own table in the database which is defined as follows:

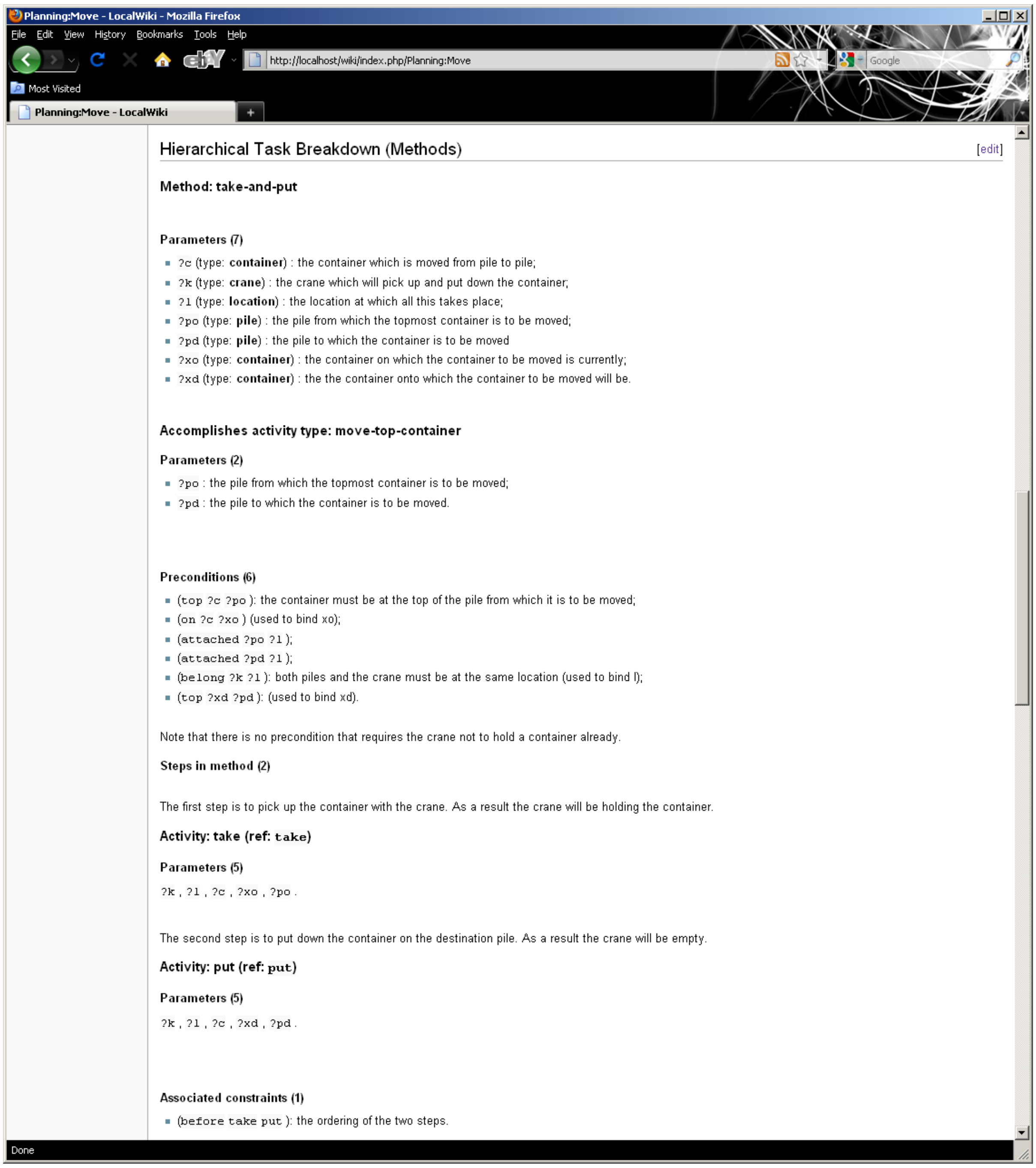
```
create table mw_plan_domain (
  aid int unsigned not null,
  did int unsigned not null,
  name varchar(64) not null,
  owntype int unsigned,
  ownid int unsigned,

  primary key (aid, did)
);
```

Again, it is the owner field in the various elements in a domain that are used to link the domain together. The only descriptive field in this table is the `name` of the domain, which is used to define planning problems.

8.6 Classical Planning Problems

Like classical planning domains, classical planning problems are simply aggregations of the concepts


Planning:Move - LocalWiki - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/wiki/index.php/Planning:Move
Most Visited
Planning:Move - LocalWiki
[edit]
Done

Hierarchical Task Breakdown (Methods)

Method: take-and-put

Parameters (7)

- `?c` (type: **container**) : the container which is moved from pile to pile;
- `?k` (type: **crane**) : the crane which will pick up and put down the container;
- `?l` (type: **location**) : the location at which all this takes place;
- `?po` (type: **pile**) : the pile from which the topmost container is to be moved;
- `?pd` (type: **pile**) : the pile to which the container is to be moved
- `?xo` (type: **container**) : the container on which the container to be moved is currently;
- `?xd` (type: **container**) : the the container onto which the container to be moved will be.

Accomplishes activity type: move-top-container

Parameters (2)

- `?po` : the pile from which the topmost container is to be moved;
- `?pd` : the pile to which the container is to be moved.

Preconditions (6)

- (`top ?c ?po`): the container must be at the top of the pile from which it is to be moved;
- (`on ?c ?xo`) (used to bind xo);
- (`attached ?po ?l`);
- (`attached ?pd ?l`);
- (`belong ?k ?l`): both piles and the crane must be at the same location (used to bind l);
- (`top ?xd ?pd`): (used to bind xd).

Note that there is no precondition that requires the crane not to hold a container already.

Steps in method (2)

The first step is to pick up the container with the crane. As a result the crane will be holding the container.

Activity: take (ref: `take`)

Parameters (5)

`?k` , `?l` , `?c` , `?xo` , `?po` .

The second step is to put down the container on the destination pile. As a result the crane will be empty.

Activity: put (ref: `put`)

Parameters (5)

`?k` , `?l` , `?c` , `?xd` , `?pd` .

Associated constraints (1)

- (`before take put`): the ordering of the two steps.

Figure 9: A Refinement/Method rendered in the Wiki

listed above. A planning problem contains a reference to a domain, an initial state, and a goal. The domain is simply an attribute of the problem and uses the name defined in the domain definition. Hence, domain names must be unique even across articles. The initial state and the goal are both XML elements defined in the extension, each grouping a set of atoms together. Since state descriptions tend to be rather lengthy, the following only lists the beginning of the definition of an initial state:

```
<state ref="dwr-problem1">

First, there are the static relations
which describe the topology of the world
and the immobile objects which exist in it.
In short, there are two adjacent location
with a crane and two piles each.

There are two locations that are adjacent
to each other. Note that the symmetry of
the adjacency relation has to be made
explicit:

* <atom>
  <rel name="adjacent" />
  <object name="l1" />
  <object name="l2" />
</atom>;
* <atom>
  <rel name="adjacent" />
  <object name="l2" />
  <object name="l1" />
</atom>;

There is one crane at each of the two
locations:

* <atom>
  <rel name="belong" />
  <object name="k1" />
  <object name="l1" />
</atom>;
* <atom>
  <rel name="belong" />
  <object name="k2" />
  <object name="l2" />
</atom>;
...
```

The `state` element contains all the atoms the define the world state. An optional attribute allows to name this state such that it may be re-used in a different context, a feature which is not used at present. In a classical world state all atoms must be positive, and they are simply listed here, together with plenty of commenting text. The database table for world states is defined as follows:

```
create table mw_plan_worldstate (
  aid int unsigned not null,
  sid int unsigned not null,
  uref varchar(64),
  owntype int unsigned,
  ownid int unsigned,

  primary key (aid, sid)
);
```

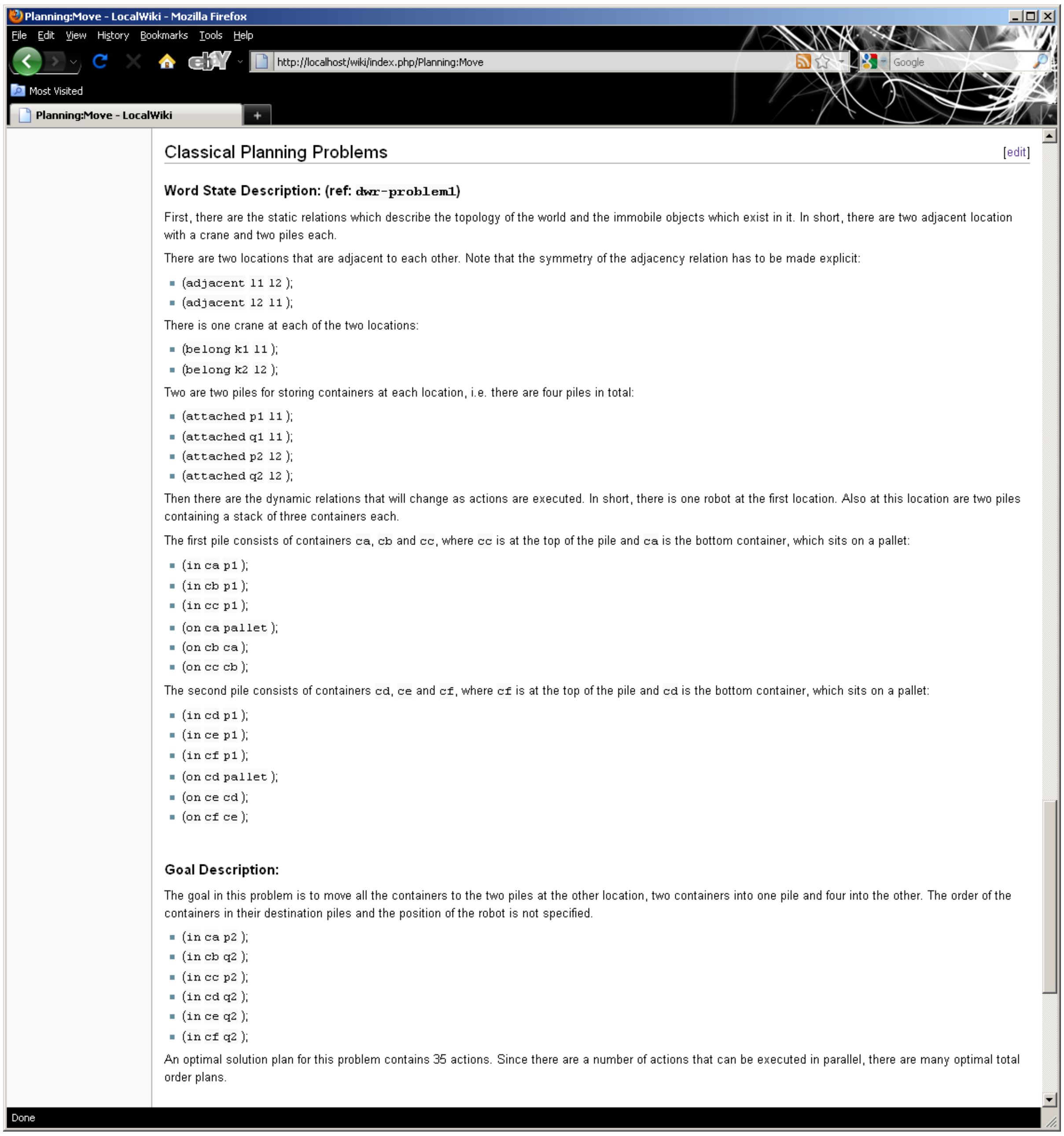
As before, the way atoms are groups is stored within the atoms, which are in the conditions table, where the owner is the same world state. Goals are almost identical in terms of their representation, which is why no example is given here. The only difference is that atom in goals may be negated, but this feature is not used in the example used. The page that is generated by MediaWiki which contains the state (partially) listed above is shown in figure 10.

8.7 Beliefs, Desires and Intentions

The mental attitudes of the BDI paradigm are again relatively simple structures that are not very interesting from a syntactical point of view. Beliefs are currently only set of atoms (like in a world state) that are associated with an agent. More complex beliefs may be introduced as required. Desires and intentions are syntactically identical in that they relate an activity (we adopt a task-centric view) to an agent.

The database schema for beliefs is defined as follows:

```
create table mw_plan_belief (
  aid int unsigned not null,
  bid int unsigned not null,
  agent varchar(64) not null,
  tpt datetime not null,
```


Planning:Move - LocalWiki - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/wiki/index.php/Planning:Move
Google
Most Visited
Planning:Move - LocalWiki
[edit]
Done

Classical Planning Problems

Word State Description: (ref: `dwr-problem1`)

First, there are the static relations which describe the topology of the world and the immobile objects which exist in it. In short, there are two adjacent location with a crane and two piles each.

There are two locations that are adjacent to each other. Note that the symmetry of the adjacency relation has to be made explicit:

- (`adjacent l1 l2`);
- (`adjacent l2 l1`);

There is one crane at each of the two locations:

- (`belong k1 l1`);
- (`belong k2 l2`);

Two are two piles for storing containers at each location, i.e. there are four piles in total:

- (`attached p1 l1`);
- (`attached q1 l1`);
- (`attached p2 l2`);
- (`attached q2 l2`);

Then there are the dynamic relations that will change as actions are executed. In short, there is one robot at the first location. Also at this location are two piles containing a stack of three containers each.

The first pile consists of containers `ca`, `cb` and `cc`, where `cc` is at the top of the pile and `ca` is the bottom container, which sits on a pallet:

- (`in ca p1`);
- (`in cb p1`);
- (`in cc p1`);
- (`on ca pallet`);
- (`on cb ca`);
- (`on cc cb`);

The second pile consists of containers `cd`, `ce` and `cf`, where `cf` is at the top of the pile and `cd` is the bottom container, which sits on a pallet:

- (`in cd p1`);
- (`in ce p1`);
- (`in cf p1`);
- (`on cd pallet`);
- (`on ce cd`);
- (`on cf ce`);

Goal Description:

The goal in this problem is to move all the containers to the two piles at the other location, two containers into one pile and four into the other. The order of the containers in their destination piles and the position of the robot is not specified.

- (`in ca p2`);
- (`in cb q2`);
- (`in cc p2`);
- (`in cd q2`);
- (`in ce q2`);
- (`in cf q2`);

An optimal solution plan for this problem contains 35 actions. Since there are a number of actions that can be executed in parallel, there are many optimal total order plans.

Figure 10: A Planning Problem rendered in the Wiki

```
  owntype int unsigned,
  ownid int unsigned,

  primary key (aid, bid)
);
```

Apart from the agent that holds the belief, this may also include a time point at which the agent holds this belief. This is a compromise between having no times associated with beliefs and having intervals associated with them. Whether this is sufficient really depends on the reasoning engine that will use this information. Desires and intentions are very similar and they are omitted here.

8.8 Agent Capabilities

The final concept to be described here are agent capabilities which are used in many frameworks for multi-agent planning. Capabilities can be seen as planning operators, instances of which can be assigned to an agent that has the capability in a plan. They can also be compared to task that can be accomplished in HTN planning. In both cases, the representation consists of a symbol representing the name of the activity that is accomplished with some parameters representing the objects involved in applying the capability. An example of a capability representation is shown in the following:

```
<capability agent="crane456a">

<accomplishes>
<activity type="move-top-container">
<parameters>
* <var name="po" />: the pile from which
  the topmost container is to be moved;
* <var name="pd" />: the pile to which
  the container is to be moved.
</parameters>
</activity>
</accomplishes>

<constraints>
* <atom>
  <rel name="attached" />
  <var name="po" />
  <object name="loc456" />
</atom>;
* <atom>
  <rel name="attached" />
  <var name="pd" />
  <object name="loc456" />
  </atom>;
</constraints>

</capability>
```

In addition to the activity accomplished, a capability description may contain constraints on the applicability of the capability. These are similar to the preconditions associated with a method. In fact, syntactically they are the same. Currently these conditions are limited to static relations though, i.e. relations that do not change over time, such as the topology of the locations. This means capabilities are either applicable, or they are not. there is no way to make them applicable in a different situation. The intention is still to have agents have autonomy and be able to decide whether they can apply a given capability, depending on many factors. The constraints can be used to filter out many plans as infeasible though, which saves planning effort.

The database table that holds the capabilities is defined as follows:

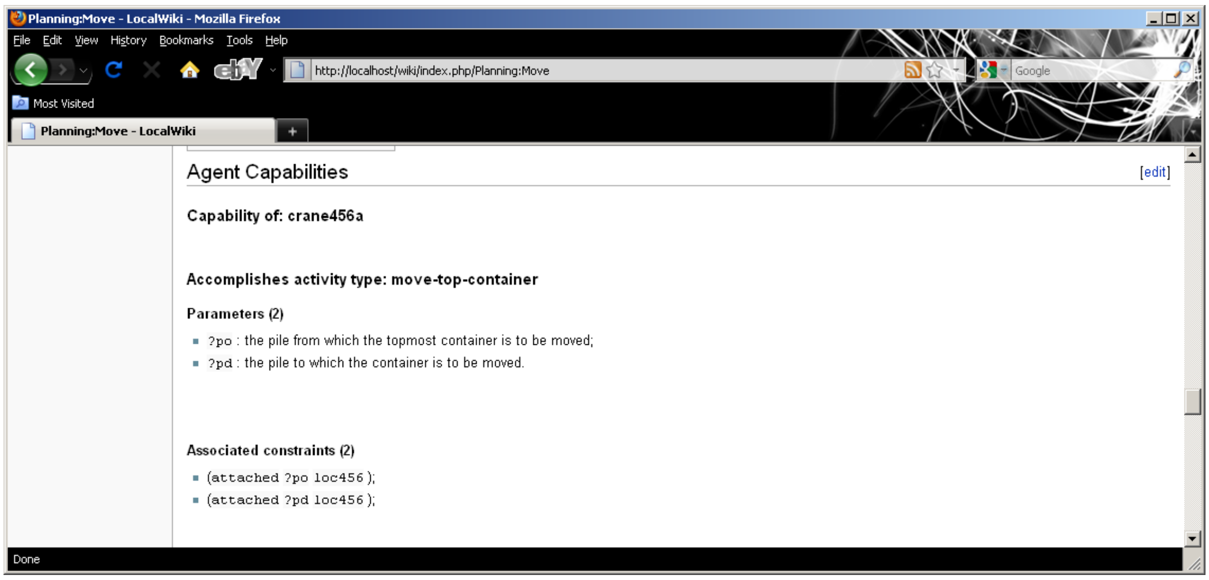
```
create table mw_plan_capability (
  aid int unsigned not null,
  cid int unsigned not null,
  agent varchar(64) not null,
  task varchar(255) not null,
  owntype int unsigned,
  ownid int unsigned,

  primary key (aid, cid)
);
```

The constraints on a capability are again in the conditions and have the capability as their owner. The capability defined in the example above is rendered in figure 11.

9 Domain Features to Facilitate Knowledge Engineering

Specifying a planning domain and a planning problem in a formal description language defines a



Planning:Move - LocalWiki - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/wiki/index.php/Planning:Move
Most Visited
Planning:Move - LocalWiki
[edit]
Done

Agent Capabilities

Capability of: crane456a

Accomplishes activity type: move-top-container

Parameters (2)

- ?po : the pile from which the topmost container is to be moved;
- ?pd : the pile to which the container is to be moved.

Associated constraints (2)

- (attached ?po loc456);
- (attached ?pd loc456);

Figure 11: A Capability rendered in the Wiki

search space that can be traversed by a state-space planner to find a solution plan. It is well known that this specification process, also known as *problem formulation* [Russell and Norvig, 2003], is essential for enabling efficient problem-solving though search [Amarel, 1968].

The Planning Domain Definition Language (PDDL) [Fox and Long, 2003] has become a de-facto standard for specifying STRIPS-like planning domains and problems with various extensions. PDDL allows for the specification of some auxiliary information about a domain, such as types, but this information is optional.

9.1 Domain Features

In this paper we will formally define four domain features that can be used to assist knowledge engineers during the problem formulation process, i.e. the authoring of a planning domain which defines the state space. These features may also be exploited by a planning algorithm to speed up the search, but this possibility depends on the actual planning algorithm used and will not be evaluated in this paper. The features defined here are: *domain types*, *relation fluency*, *inconsistent effects* and *reversible actions*. These features are not new, at least at an informal level. Their specification is either already part of PDDL or could easily be added to the language.

The values these features take for a given domain can also be computed independent of their explicit specification. A comparison of the computed features to the ones specified in the formal domain definition can then be used to validate the formalization, thus supporting the domain author in producing a consistent domain. Applying this approach to various planning domains shows that the features defined here can be used to identify certain representational problems.

9.2 Related Work

Amongst the features mentioned above, domain types have been discussed most in the planning literature. A rigorous method for problem formulation in the case of planning domains was presented in [McCluskey and Porteous, 1997]. In the second step of their methodology types are extracted from an informal description of a planning domain. Types have been used as a basic domain feature in TIM [Fox and Long, 1998]. Their approach exploits functional equivalence of objects to derive a hierarchical type structure. The difference between this approach and our algorithm will be explained in the relevant section below. This work has later been extended to infer generic types such as mobiles and resources that can be exploited to optimize plan search [Coles and Smith, 2006].

The distinction between rigid and fluent relations [Ghallab *et al.*, 2004] is common in AI planning and will be discussed only briefly. Inconsistent effects of different actions are exploited in the GraphPlan algorithm [Blum and Furst, 1995] to define the mutex relation. However, this is applied to pairs of actions (i.e. fully ground instances of operators) rather than operators. Reversible actions, as a domain feature, are not related to regression of goals, meaning this feature is unrelated to the direction of search (forward from the initial state or regressing backwards from the goal). The reversibility of actions (or operators) does not appear to feature much in the AI planning literature. However, in generic search problems they are a common technique used to prune search trees [Russell and Norvig, 2003].

Preprocessing of planning domains is a technique that has been used to speed up the planning process [Dawson and Siklossy, 1977]. Perhaps the most common preprocessing step is the translation of the STRIPS (function-free, first-order) representation into a propositional representation. An informal algorithm for this is described in [Ghallab *et al.*, 2004, section 2.6]. A conceptual flaw in this algorithm (highlighted by the analysis of inconsistent effects) will be briefly discussed in the conclusions of this paper.

10 Type Information

Many planning domains include explicit type information. In PDDL the `:typing` requirement allows the specification of typed variables in predicate and operator declarations. In problem specifications, it allows the assignment of constants or objects to types. If nothing else, typing tends to greatly increase the readability of a planning domain. However, it is not necessary for most planning algorithms to work.

In this section we will show how type information can be inferred from the operator descriptions in the planning domain definition. If the planning domain includes explicit type information the inferred types can be used to perform a consistency check, thus functioning as a knowledge engineering tool. In any case, type information can be used to simplify parts of the planning process. For example, if the planner needs to propositionalize the planning domain, type information can be used to limit the number of possible values for variables, or a ground backward searcher may use this information to similar effect.

The formalism that follows is necessary to show that the derived type system is **maximally specific** given the knowledge provided by the operators, that is, any type system that further subdivides a derived type must necessarily lead to a search space that contains type inconsistent states.

10.1 Type Consistency

The simplest kind of type system often used in planning is one in which the set of all constants C used in the planning domain and problem is divided into disjoint types T. That is, each type corresponds to a subset of all constants and each constant belongs to exactly one type. This is the kind of type system we will look at here.

Definition 1 (type partition) *A* ***type partition*** $\mathcal{P}$ *is a tuple* $\langle C, T, \tau \rangle$ *where:*

- C *is a finite set of* $n(C) \geq 1$ *constant symbols* $C = \{c_1, \ldots, c_{n(C)}\}$,
- T *is a set of* $n(T) \leq n(C)$ *types* $T = \{t_1, \ldots, t_{n(T)}\}$, *and*
- $\tau : C \to T$ *is a function defining the type of a given constant.*

A type partition divides the set of all constants that may occur in a planning problem into a set of equivalence classes. The availability of a type partition can be used to limit the space of world states that may be searched by a planner. In general, a world state in a planning domain can be any subset of the powerset of the set of ground atoms over predicates P with arguments from C.

Definition 2 (type function) *Let* $P = \{P_1, \ldots, P_{n(P)}\}$ *be a set of* $n(P)$ *predicate symbols with associated arities* $a(P_i)$ *and let* $T = \{t_1, \ldots, t_{n(T)}\}$ *be a set of types. A* ***type function*** *for predicates is a function*

$$arg_P : P \times \mathbb{N} \to T$$

which, for a given predicate symbol P_i and argument number $1 \leq k \leq a(P_i)$ gives the type $arg_P(P_i, k) \in T$ of that argument position.

This is the kind of type specification we find in PDDL domain definitions as part of the definition of predicates used in the domain, provided that the typing extension of PDDL is used. The type function is defined by enumerating the types for all the arguments of each predicate.

Definition 3 (type consistency) *Let $\langle C, T, \tau\rangle$ be a type partition. Let $P_i \in P$ be a predicate symbol and let $c_1, \ldots, c_{a(P_i)} \in C$ be constant symbols. The ground first-order atom $P_i(c_1, \ldots, c_{a(P_i)})$ is **type consistent** iff $\tau(c_k) = arg_P(P_i, k)$. A world state is **type consistent** iff all its members are type consistent.*

Thus, for a given predicate P_i there are $|C|^{a(P_i)}$ possible ground instances that may occur in world states. Clearly, the set of type consistent world states is a subset of the set of all world states. The availability of a set of types can also be used to limit the actions considered by a planner.

Definition 4 (type function) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of $n(O)$ operator names with associated arities $a(O_i)$ and let $T = \{t_1, \ldots, t_{n(T)}\}$ be a set of types. A **type function** for operators is a function*

$$arg_O : O \times \mathbb{N} \to T$$

which, for a given operator symbol O_i and argument number $1 \leq k \leq a(O_i)$ gives the type $arg_O(O_i, k) \in T$ of that argument position.

Again, this is exactly the kind of type specification that may be provided in PDDL where the function is defined by enumeration of all the arguments with their types for each operator definition.

Definition 5 (type consistency) *Let $\langle C, T, \tau\rangle$ be a type partition. Let $O_i(v_1, \ldots, v_{a(O_i)})$ be a STRIPS operator defined over variables $v_1, \ldots, v_{a(O_i)}$ with preconditions $precs(O_i)$ and effects $effects(O_i)$, where each precondition/effect has the form $P_j(v_{P_j,1}, \ldots, v_{P_j,a(P_j)})$ or $\neg P_j(v_{P_j,1}, \ldots, v_{P_j,a(P_j)})$ for some predicate $P_j \in P$. The operator O_i is **type consistent** iff:*

- *all the operator variables $v_1, \ldots, v_{a(O_i)}$ are mentioned in the positive preconditions of the operator, and*
- *if $v_k = v_{P_j,l}$, i.e. the kth argument variable of the operator is the same as the lth argument variable of a precondition or effect, then the types must also be the same: $arg_O(O_i, k) = arg_P(P_j, l)$.*

The first condition is often required only implicitly (see [Ghallab *et al.*, 2004, chapter 4]) to avoid the complication of "lifted" search in forward search. We will use this condition shortly to show that a type consistent system is closed.

Given a type partition $\langle C, T, \tau\rangle$ and type functions arg_P and arg_O, we can define a most general state-transition system over all type consistent states as follows:

Definition 6 (state-transition system Σ^*) *Let $\langle C, T, \tau\rangle$ be a type partition. Let $P = \{P_1, \ldots, P_{n(P)}\}$ be a set of predicate symbols with associated type function arg_P and let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of type consistent operators. Then $\Sigma^* = (S^*, A^*, \gamma)$ is a (restricted) state-transition system, where:*

- *S^* is the powerset of the set of all type consistent ground atoms with predicates from P and arguments from C,*
- *A^* is the set of all (type consistent) ground instances of operators from O, and*
- *γ is the usual state transition function for STRIPS actions: $\gamma(s, a) = (s - effects^-(a)) \cup effects^+(a)$ iff action a is applicable in state s*[1].

This state-transition system forms a supersystem to a state-transition system defined by a planning problem containing a type consistent initial state, and a set of type consistent operator definitions, in the sense that the states of that system (the reachable states from the initial states) must be a subset of S^* and the actions must be a subset A^*. It is therefore interesting to observe that Σ^* is closed:

Proposition 1 (closed Σ^*) *Let $s \in S^*$ be a type consistent state, i.e. a type consistent set of ground atoms. Let $a \in A^*$ be a type consistent action that*

[1]See the definition of a STRIPS operator in [Ghallab *et al.*, 2004, page 28] and the discussion of inconsistent effects below.

is applicable in s. Then the successor state $\gamma(s,a)$ is a type consistent state in S^*.

To show that the above is true, we need to show that every atom in $\gamma(s,a)$ is type consistent. Each atom in $\gamma(s,a)$ was either in the previous state, s, in which case it was type consistent by definition, or it was added as a positive effect. Since the action is an applicable instance of a type consistent operator O_i there must be a substitution σ such that $\sigma(\mathit{precs}^+(O_i)) \subseteq s$. Furthermore, this substitution grounds every operator variable because type consistency requires all of them to occur in the positive preconditions. Given the type consistency of s, all arguments in $\sigma(\mathit{precs}^+(O_i))$ must agree with arg_P. Given the type consistency of O_i, all arguments of a must agree with arg_O, and therefore so must the effects $\sigma(\mathit{effects}(O_i))$. Hence, all positive effects are type consistent, meaning every element of $\gamma(s,a)$ must be type consistent. ■

10.2 Derived Types

The above definitions assume that there is an underlying type system that has been used to define the planning domain and problems in a consistent fashion. We shall continue to assume that such a type system exists, but it may not have been explicitly specified in the PDDL definition of the domain. We shall now define a type system that is derived from the operator descriptions in the planning domain.

Definition 7 (type name) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of STRIPS operators. Let P be the set of all the predicate symbols used in all the operators. A* ***type name*** *is a pair $\langle N, k\rangle \in (P \cup O) \times \mathbb{N}$.*

A type name can be used to refer to a type in a derived type system. There usually are multiple names to refer to the same type. The basic idea behind the derived types is to partition the set of all type names into equivalence classes, and then assign constants used in a planning problem to different equivalence classes, thus treating each equivalence class as a type.

Definition 8 (O-type) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of STRIPS operators over operator variables $v_1, \ldots, v_{a(O_i)}$ with $\mathit{conds}(O_i) = \mathit{precs}(O_i) \cup \mathit{effects}(O_i)$ and all operator variables mentioned in the positive preconditions. Let P be the set of all the predicate symbols used in all the operators. An* ***O-type*** *is a set of type names. Two type names $\langle N_1, i_1\rangle$ and $\langle N_2, i_2\rangle$ are in the same O-type, denoted $\langle N_1, i_1\rangle \equiv_O \langle N_2, i_2\rangle$, iff one of the following holds:*

- *$N_1(v_{1,1}, \ldots, v_{1,a(N_1)})$ is an operator with precondition or effect $N_2(v_{2,1}, \ldots, v_{2,a(N_2)}) \in \mathit{conds}(N_1)$ which share a specific variable: $v_{1,i_1} = v_{2,i_2}$,*
- *$N_2(v_{2,1}, \ldots, v_{2,a(N_2)})$ is an operator with precondition or effect $N_1(v_{1,1}, \ldots, v_{1,a(N_1)}) \in \mathit{conds}(N_2)$ which share a specific variable: $v_{1,i_1} = v_{2,i_2}$, or*
- *there is a type name $\langle N, j\rangle$ such that $\langle N, j\rangle \equiv_O \langle N_1, i_1\rangle$ and $\langle N, j\rangle \equiv_O \langle N_2, i_2\rangle$.*

Definition 9 (O-type partition) *Let (s_i, g, O) be a STRIPS planning problem. Let C be the set of all constants used in s_i. Let $T = \{t_1, \ldots, t_{n(T)}\}$ be the set of O-types derived from the operators in O. Then we can define the function $\tau : C \to T$ as follows:*

$$\tau(c) = t_i : \forall R(c_1, \ldots, c_{a(R)}) \in s_i : \\ (c_j = c) \Rightarrow \langle R, j\rangle \in t_i$$

Note that $\tau(c)$ is not necessarily well-defined for every constant mentioned in the initial state, e.g. if a constant is used in two relations that would indicate different derived types (which rely only on the operator descriptions). In this case the O-type partition cannot be used as defined above. However, if appropriate unions of O-types are taken then this results in a new type partition for which $\tau(c)$ is defined. In the worst case this will lead to a type partition consisting of a single type. Given that this approach is always possible, we shall now assume that $\tau(c)$ is always defined.

Definition 10 *Let $T = \{t_1, \ldots, t_{n(T)}\}$ be the set of O-types for a given set of operators O and let $P = \{P_1, \ldots, P_{n(P)}\}$ be the predicates that occur on operators from O. We can easily define type functions arg_P and arg_O as follows:*

$$\mathit{arg}_P(P_i, k) = t_i : \langle P_i, k\rangle \in t_i \text{ and} \\ \mathit{arg}_O(O_i, k) = t_i : \langle O_i, k\rangle \in t_i$$

Proposition 2 *Let (s_i, g, O) be a* STRIPS *planning problem and let $\langle C, T, \tau\rangle$ be the O-type partition derived from this problem. Then every state that is reachable from the initial state s_i is type consistent.*

To show this we first show that the initial state is type consistent. Since the definition of τ is based on the argument positions in which they occur in the initial state, this follows trivially.

Next we need to show that every action that is an instance of an operator in O is type consistent. All operator variables must be mentioned in the positive preconditions according to the definition of an O-type. Furthermore, if a precondition or effect share a variable with the operator, these must have the same type since $\equiv_O$ puts them into the same equivalence class.

Finally we can show that, if action a is applicable in a type consistent state s, the resulting state $\gamma(s, a)$ must also be type consistent. Every atom must come either from s in which case it must be type consistent, or it comes from a positive effect, which, given the type consistency of a means it must also be type consistent. ■

This shows that the type system derived from the operator definitions is indeed useful as it creates a state space of type consistent states. However, the question that remains is whether it is the best or even only type system. Clearly, there may be other type systems that give us type consistent state space. The system that consists just of a single type is a trivial example. A better type system would divide the set of constants into more types though, as this reduces the size of a type consistent state space. We will now show that the above type system is maximally specific given the knowledge provided by the operators.

Theorem 1 *Let (s_i, g, O) be a* STRIPS *planning problem and let $\langle C, T, \tau\rangle$ be the O-type partition derived from this problem. If two constants c_1 and c_2 have the same type $\tau(c_1) = \tau(c_2)$ then they must have the same type in every type partition that creates a type consistent search space.*

The first step towards showing that the above holds is the insight that operators can be used to constrain types in both directions, forward and backward. If an operator variable v_i appears in a precondition and an effect, then the type of the position of the predicate in the effect must be subset of the type of the position in the precondition or the application of the operator may lead to a state that is not type consistent. Since types are defined by an equivalence relation, however, the two types must actually be the same type. Hence the type in the effect also constrains the type in the precondition.

Now, for two type names to be in the same O-type, there must be a connecting chain $\langle R_0 O_1 R_1 \ldots O_n R_n\rangle$ of alternating first order literals and operators such that R_{i-1} and R_i are conditions of O_i which share an operator variable as the j_{i-1}th and j_ith argument respectively. The variable that is shared may vary along the chain. For each step along the chain, if a constant may occur in the j_{i-1}th position in R_{i-1} it may also occur in the j_ith position in R_i. Thus, there may be two type consistent states that are connected by O_i and which contain instances of R_{i-1} and R_i. Since both states are type consistent, both instances must be type consistent, too.

Now let us assume that c_1 appears as j_0th argument in R_0 and let c_2 appears as j_nth argument in R_n. Furthermore, let us assume these exists a type partition that assigns c_1 and c_2 to different types. Since c_1 is the j_0th argument in R_0 there may be another state in which c_1 appears as j_nth argument in R_n. Thus it appears in the same position of the same predicate as c_2, which means it must have the same type to be type consistent. ■

10.3 An Efficient Algorithm

The algorithm to derive domain types t_d treats types as sets of predicate and argument-number pairs. That is $t_d \subseteq 2^{P\times\mathbb{N}}$. Each domain type t_d corresponds to exactly one type $t \in T$. The only argument taken by the algorithm is the set of operator definitions O.

```
**function** extract-types($O$)
    $pTypes \leftarrow \emptyset$
    $vTypes \leftarrow \emptyset$
    **for every** $op \in O$ **do**
        extract-types($op, pTypes, vTypes$)
    **return** $pTypes$
```

The variable *pTypes* contains the O-types that have been discovered so far. Initially there are no

O-types and the set is empty. *vTypes* is a set of pairs of variables (used in operator definitions) and O-types, best implemented as a map and also initially empty. The procedure then analyzes each operator in the given set, thereby building up the type system incrementally.

```
**function** extract-types($op, pTypes, vTypes$)
  **for every** $p \in pre(op) \cup eff(op)$ **do**
    **for** $i = 1$ **to** $a(p)$ **do**
      $t_{pi} \leftarrow t_d \in pTypes : \langle rel(p), i\rangle \in t_d$
      $\langle v, t_v\rangle \leftarrow v_t \in vTypes : \exists t_d : v_t = \langle arg(i,p), t_d\rangle$
      **if** $undef(\langle v, t_v\rangle)$ **do**
        **if** $undef(t_{pi})$ **do**
          $t_{pi} \leftarrow \{\langle rel(p), i\rangle\}$
          $pTypes \leftarrow pTypes \cup t_{pi}$
        $vTypes \leftarrow vTypes \cup \langle arg(i,p), t_{pi}\rangle$
      **else**
        **if** $undef(t_{pi})$ **do**
          $t_v \leftarrow t_v \cup \{\langle rel(p), i\rangle\}$
        **else**
          merge-types($t_v, t_{pi}, pTypes, vTypes$)
```

The analysis of a given operator goes through every precondition and effect of the operator, looking at every argument position in turn. The next steps of the algorithm depend on whether the predicate-position combination has been used before (in which case it will appear in the *pTypes*) and whether the variable at that position has been used before (in which case it will be a key in the *vTypes*). If only one or neither have been used, the algorithm simply adds the relevant elements to the *pTypes* and the *vTypes*. If both have been used it may be necessary to merge the respective O-types.

```
**function** merge-types($t_1, t_2, pTypes, vTypes$)
    **if** $t_1 = t_2$ **do**
        **return**
    $pTypes \leftarrow pTypes - \{t_1, t_2\}$
    $t_{new} \leftarrow t_1 \cup t_2$
    $pTypes \leftarrow pTypes \cup \{t_{new}\}$
    **for every** $\langle v, t_v\rangle \in vTypes$ **do**
        **if** $(t_v = t_1) \vee (t_v = t_2)$ **do**
            $vTypes \leftarrow vTypes - \langle v, t_v\rangle$
            $vTypes \leftarrow vTypes + \langle v, t_{new}\rangle$
```

Of course, no action is required if the type of the variable and the type for the predicate-position combination is the same. Otherwise we replace the two sets representing the (previously different) types in *pTypes* with a new type that is the union of the two sets. Also we need to update the pairs in *vTypes* to ensure that keys that previously had one of the now removed types as value will now get the new type as their new value.

It is easy to see that the algorithm runs in polynomial time. Furthermore, the analysis performed by the algorithm uses only the operator descriptions, and thus its run time does not depend on the problem size.

This algorithm shares the input with TIM [Fox and Long, 1998], namely the operator specifications. Both algorithms use the argument positions in which parameters occur in preconditions and effects as the basis for their analysis. TIM uses this information to construct a set of finite state machines to model transitions of objects, whereas our algorithm builds the equivalence classes directly. The result produced by TIM is a hierarchical type system that is used to derive state invariants. In contrast, the type system derived by our algorithm is flat, meaning it may be less discriminating than the structure derived by TIM. However, we could show that the types derived by our algorithm are maximally specific for given operator descriptions. In addition, a flat type system can be used to enrich the operator definitions explicitly by simply adding unary predicates as type preconditions.

10.4 Evaluation

To evaluate the algorithm we have applied it to a small number of planning domains. To avoid any bias we used only planning domains that were available from third parties, mostly from the international planning competition. Since the algorithm works on domains and the results have to be interpreted manually only a limited number of experiments was possible. Random domains are not suitable as they cannot be expected to encode an implicit type system. The algorithm has been used on random domains, but this did not result in any useful insights.

A planning domain on which the algorithm has been used is the DWR domain [Ghallab *et al.*, 2004]. In this domain types are defined explicitly, so it was possible to verify consistency with the given types. The algorithm produced the following, listing the argument positions in predicates where they are used (the *pTypes*):

```
type: [loaded-0, unloaded-0, at-0]
type: [attached-0, top-1, in-1]
type: [occupied-0, attached-1, belong-1,
  adjacent-1, adjacent-0, at-1]
type: [belong-0, holding-0, empty-0]
type: [loaded-1, holding-1, on-1, on-0,
  in-0, top-0]
```

The first type states that it is used as the first argument in the `loaded`, `unloaded` and `at` predicate. This corresponds exactly to the `robot` type in the PDDL specification of the domain. Similarly, the other types correspond to `pile`, `location`, `crane` and `container`, in this order. The main difference is that the derived types do not have intelligible names.

The other domains that were used for testing did not come with type information specified in the same way as the DWR domain. However, they all use unary predicates to add type information to the preconditions (but not every unary predicate is a type). The domains used are the following STRIPS domains from the international planning competition: `movie`, `gripper`, `logistics`, `mystery`, `mprime` and `grid`. The algorithm derives between 3 and 5 types for each of these domains which appears consistent with what the domain authors had in mind. The only domain that stands out is the first, in which each predicate has its own type. However this appears to be appropriate for this very simple domain.

11 Static and Fluent Relations

Another domain feature that is useful for the analysis of planning domains concerns the relations that are used in the definition of the operators. The set of predicates used here can be divided into static (or rigid) relations and fluent (or dynamic) relations, depending on whether atoms using this predicate can change their truth value from state to state.

Definition 11 (static/fluent relation) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of operators and let $P = \{P_1, \ldots, P_{n(P)}\}$ be a set of all the predicate symbols that occur in these operators. A predicate $P_i \in P$ is **fluent** iff there is an operator $O_j \in O$ that has an effect that uses the predicate P_i. Otherwise the predicate is **static**.*

The algorithm for computing the sets of fluent and static predicate symbols is trivial and hence, we will not list it here.

There are at least two ways in which this information can be used in the validation of planning problems. Firstly, if the domain definition language allowed the domain author to specify whether a relation is static or fluent then this could be verified when the domain is parsed. This might highlight problems with the domain. Secondly, in a planning problem that uses additional relations these could be highlighted or simply removed from the initial state.

The computation of static and fluent relations has been tested on the same domains as the derived types. As is to be expected, nothing interesting can be learned from this experiment.

12 Inconsistent Effects

In a STRIPS-style operator definition the effects are specified as and add- and delete-lists consisting of a set of (function-free) first-order atoms, or a set of first-order literals where positive elements correspond to the add-list and negative elements correspond to the delete-list. Normally, the definition of an operator permits potentially inconsistent effects, i.e. a positive and a negative effect may be complementary.

12.1 Operators

Definition 12 (potential inconsistency) *Let O be a planning operator with positive effects $e^p_1, \ldots, e^p_{n(e^p)}$ and negative effects $e^n_1, \ldots, e^n_{n(e^n)}$, where each positive/negative effect is a first-order atom. O has **potentially inconsistent effects** iff O has a positive effect e^p_i and a negative effect e^n_j for which there exists a substitution σ such that $\sigma(e^p_i) = \sigma(e^n_j)$.*

It is fairly common for planning domains to define operators with potentially inconsistent effects.

For example, the move operator in the DWR domain is defined as follows:

```
(:action move
  :parameters (?r ?fr ?to)
  :precondition (and (adjacent ?fr ?to)
    (at ?r ?fr) (not (occupied ?to)))
  :effect (and (at ?r ?to) (occupied ?to)
    (not (occupied ?fr)) (not (at ?r ?fr))))
```

This operator has a positive effect `(at ?r ?to)` and a negative effect `(at ?r ?fr)`. These two effects are unifiable and represent a potential inconsistency. Since this is a common feature in planning domains there is no need to raise this to the domain author. Effects that are necessarily inconsistent may be more critical.

Definition 13 (necessary inconsistency) *Let O be a planning operator with positive effects $E^p = \{e^p_1, \ldots, e^p_{n(e^p)}\}$ and negative effects $E^n = \{e^n_1, \ldots, e^n_{n(e^n)}\}$, where each positive/negative effect is a first-order atom. O has* ***necessarily inconsistent effects*** *iff O has a positive effect e^p_i and a negative effect e^n_j such that $e^p_i = e^n_j$.*

None of the domains used in the experiments above specified an operator with necessarily inconsistent effects. Given the definition of the state-transition function for STRIPS operators [Ghallab *et al.*, 2004] as

$$\gamma(s,a) = (s - E^n) \cup E^p$$

it should be clear that the negative effect e^n_j can be omitted from the operator description without changing the set of reachable states. If $e^n_j \notin s$ then its removal from s will not change s, and the addition of e^p_i ensures that $e^n_j \in \gamma(s,a)$ because $e^p_i = e^n_j$. If $e^n_j \in s$ it will be removed in $\gamma(s,a)$, but it will subsequently be re-added. Thus, the presence of the negative effect does not change the range of the state-transition function.

From a knowledge engineering perspective this means that an operator with necessarily inconsistent effects indicates a problem and should be raised to the domain author. However, this is only true for simple STRIPS operators where actions are instantaneous and thus, all effects happen simultaneously. If effects are permitted at different time points then only those that are necessarily inconsistent at the same time point must be considered a problem.

12.2 Actions

Since actions are ground instances of operators, there is no need to distinguish between necessarily and potentially inconsistent effects. All effects must be ground for actions and therefore inconsistent effects are always necessarily inconsistent. Even if necessarily inconsistent operators are not permitted in a domain, actions with inconsistent effects may still occur as instances of operators with potentially inconsistent effects.

Whether it is desirable for the planner to consider such actions depends on the other effects of the action. For example, in the DWR domain no action with inconsistent effects needs to be considered. However, if an action has side effects then it may make sense to permit such actions. For example, circling an aircraft in a holding pattern does not change the location of the aircraft, but it does reduce the fuel level. If such side effects are important actions with inconsistent effects may need to be permitted. And, of course, every action has the side effect of taking up a step in a plan.

If actions with inconsistent effects are considered by the planner, this may lead to further complications. This is because the definition of the state-transition function first subtracts negative effects from a state and then adds positive effects. For actions that have no inconsistent effects this order is irrelevant. However, if actions with inconsistent effects are permitted the result may be surprising. For example, returning to the move operator in the DWR domain, this has been defined with a positive effect `(occupied ?to)` and a negative effect `(occupied ?fr)`. Thus, the action `(move r loc loc)` will result in a state in which `(occupied loc)` holds. Now suppose the domain had been defined using the predicate `free` instead of `occupied`. In this case the result of `(move r loc loc)` would result in a state in which `(free loc)` holds. This problem occurs only with inconsistent effects.

None of the domains used in the tests above require actions with inconsistent effects and thus, they can be ignored by the planner. The following algorithm can be used to find the applicable actions (without inconsistent effects) in a given state.

```
**function** addApplicables($A, o, p, \sigma, s$)
  **if** not empty($p^+$) **then**
    let $p_{next} \in p$
    **for every** $s_p \in s$ **do**
      $\sigma' \leftarrow$ unify($\sigma(p_{next}), s_p$)
      **if** valid($\sigma'$) **then**
        addApplicables($A, o, p - p_{next}, \sigma', s$)
  **else**
    **for every** $p_{next} \in p^-$ **do**
      **if** falsifies($s, \sigma(p_{next})$) **then return**
    **for every** $e_p \in$ effects$^+(o)$ **do**
      **for every** $e_n \in$ effects$^-(o)$ **do**
        **if** $e_p = e_n$ **then return**
    $A \leftarrow A + \sigma(o)$
```

The algorithm adds all instances of operator o that are applicable in state s to the set of actions A. The parameter p represents the remaining preconditions (initially empty) and a substitution σ (also initially empty) will be built up by the algorithm. It first deals with the remaining positive preconditions and uses those to construct the substitution for all the parameters of the operators. Note that we require an operator to mention all its parameters in the positive preconditions. When the positive preconditions have been tested, the algorithm checks the negative preconditions under σ which must now be fully ground. Finally, the algorithm tests for inconsistent effects by doing a pairwise comparison between positive and negative effects. This algorithm can also be used to generate the actions for the next action layer in a planning graph. A goal regression version is slightly different as it is no longer guaranteed that all the operator parameters will be bound after the unification with a goal (and possibly static preconditions).

13 Reversible Actions

A common feature in many planning domains (and in many classic search problems) is that they contain actions that can be reversed by applying another action. There is usually no need to consider such actions during the search process.

13.1 Reversible Operators

The idea here is to apply the concept of reversibility to operators: an operator may be reversed by another operator (or the same operator), possibly after a suitable substitution of variables occurring as parameters in the operator definition. Note that this definition is somewhat narrow as it demands this pattern to be consistent across all instances of the two operators, i.e. it excludes the possibility of an operator sometimes being reversed by one operator, and sometimes by another, depending on the values of the parameters.

Definition 14 (reversing operators) *An action a that is applicable in a state s is* ***reversed by an action*** *a' if the state that results from applying the sequence $\langle aa' \rangle$ in s results in s, i.e. the state remains unchanged. An operator O is* ***reversed by an operator*** *O' under substitution σ' iff for every action $a = \sigma(O)$ that is an instance of O:*

- *if a is applicable in a state s then $a' = \sigma(\sigma'(O'))$ is applicable in $\gamma(s, a)$ and*
- *$\gamma(\gamma(s, a), a') = s$.*

For example, consider the `(move ?r ?l1 ?l2)` operator from the DWR domain. This can be reversed by another move operation with different parameters, as defined by the substitution $\sigma' =$ {`?l1`←`?l2`,`?l2`←`?l1`}, i.e. `(move ?r ?l1 ?l2)` is reversed by σ'(`(move ?r ?l1 ?l2)`) =`(move ?r ?l2 ?l1)`.

While this definition captures the idea of a reversing operator, it is not very useful from a computational point of view. Another way to avoid exploring states that are the result of the application of an action followed by its reverse action is to store all states in a hash table and test whether the new state has been encountered before, an approach that is far more general than just testing for reversing actions. Computationally, it is roughly as expensive as the test suggested by the above definition. The key here is that both are state specific. A definition of reversibility that does not depend on the state in which an action is applied would be better.

From a domain author's perspective, it is often possible to specify which operators can be used to reverse another operator, as we have shown in the DWR move example above. If this information is available during search then there is no need to apply the reverse action, generate the state, and compare it to the previous state. Instead a relatively simple substitution test would suffice: $a' = \sigma(\sigma'(O'))$.

Proposition 3 *Let O_1 be an operator with positive effects E_1^p and negative effects E_1^n that is reversed by O_2 with positive effects E_2^p and negative effects E_2^n under substitution σ'. Then the two sets of positive/negative effects must cancel each other:*

$$E_1^p = \sigma'(E_2^n) \text{ and } E_1^n = \sigma'(E_2^p)$$

Suppose there is a positive effect in E_1^p that is not in $\sigma'(E_2^n)$. Now suppose an instance of O was applied in a state in which the effect in question does not already hold. The effect would then be added by the instance of O but it would not be deleted by the reversing action, and thus the original state and the state resulting from the two actions in sequence would not be the same. A similar argument holds for an effect in E_1^n that is not in $\sigma'(E_2^p)$. ■

This means we can let the domain author specify reversing operators and then use the above necessary criterion for validation. Or we could treat the above criterion as sufficient and thus exclude a portion of the search space. This may lead to an incompleteness in the search, but the domains we have used for our evaluation do not show this problem.

13.2 Unique Reversibility

In fact we have made an even stronger assumption to carry out some experiments with the domains mentioned above: we have assumed that there is at most one operator that reverses a given operator. We have then, for each domain, done a pairwise test on all the operators defined in the domain to see whether the necessary criterion holds. This resulted in discovering that the move operator can be reversed by itself with a substitution automatically derived from the operator definition, and similarly it discovered the reversibility between the take and put operators and the load and unload operators in the DWR domain.

Perhaps surprisingly, the unique reversibility was not given for all domains. The `logistics` domain contains load and unload operators for trucks and airplanes. These are specified as four distinct operators. However, in terms of their effects the two load operators and the two unload operators cannot be distinguished. The only difference lies in the preconditions where the `?truck` parameter is required to be a truck and the `?airplane` parameter is required to be an airplane.

This result can be interpreted in two ways: one could argue that the necessary condition may not be used as sufficient in this domain. Or one could argue that this domain contains redundancy that can be removed by merging the two load and unload operators, which would not change the set of reachable states in this example but means the planner has fewer actions to consider. Either way, testing for the necessary reversibility condition has highlighted this domain feature.

14 Communicating Plans: Methods, Assumptions, and Procedures

Most of the research in AI planning has focussed on algorithms for the efficient discovery of plans that solve a given planning problem. In this work we are not interested in this classical problem, but in the question how an agent framework can support the distributed execution of plans. This is closely intertwined with the classic planning problem, of course, when execution fails and the plan needs to be modified.

The remainder of this report is structured as follows. In the next section we shall give a brief overview of the two approaches that have been undertaken to provide the foundation for meaningful agent communication. Such communication is required to manage the successful sharing and execution of plans amongst a group of distributed agents. This will define the message structure, communicative acts, and interaction protocols for agent collaboration. We will then extend the protocols specifically to support distributed plan execution. This will be followed by a discussion on plan failure and how this can be handled.

15 Background: Agent Communication

Meaningful agent communication is a difficult problem that has been addressed in agent research by two major efforts: the Knowledge Sharing Effort

(KSE) and the Foundation for Intelligent Physical Agents (FIPA). Both approaches rely on the existence of a transport layer that allows the exchange of messages between agents. At this layer a message is a stream of bytes that have no pre-defined meaning. This is sufficient for many applications in which the software developer knows the meaning of these bytes and the correct behavior can be hard-coded into the software. Such software would normally not be considered an agent. For software to be considered an agent it should be capable of communicating in a semantically rich, that is, a meaningful language, and for an agent to be capable of meaningful communication, the meaning has to exist not only in the software developer's head, but it has to be somehow encoded in the system of communicating agents.

The first step towards encoding meaning in an agent communication language usually consists of the definition of an ontology that defines the terms that can be used for communication. The symbols used to represent these terms have meaning because they are related to other terms through a set of pre-defined relations that constrain the possible interpretations. Thus, an ontology that contains only one symbol has no meaning as it is entirely unconstrained. On the other hand an ontology that contains many symbols and relations between them does encode meaning for these symbols. For agent communication, if two agents refer to the same ontology, it can be assumed that they are using the symbols in their messages with the same meaning. An ontology can be seen as providing the vocabulary for meaningful communication and the conceptual framework described in the first report constitutes part of such an ontology, namely one focussed on plans.

A shared ontology is necessary, but not sufficient for meaningful agent communication. To form meaningful statements it is necessary to define a set of grammatical rules that define how the symbols from an ontology can be combined to form more complex structures. Furthermore, a formal semantics is needed to define what exactly it means to put certain symbols together in a certain way. Together, syntax and semantics define a formal language that can be used for meaningful agent communication. While an ontology is always finite, such a language usually allows for an infinite number of statements to be formed, making it possible to express an infinite number of facts, or plans.

An ontology and a formal language are still not sufficient for meaningful agent communication though. Meaningful agent communication also requires agent messages to encode modalities. For example, an agent might have a fact in its knowledge base and it can send this as a message to another agent, but what is the receiving agent meant to do with this message. Is the sender telling it about a fact it believes, or is it asking whether the receiving agent believes the content of the message. If the content is a plan, is the sender asking the receiver to execute the plan, or should it evaluate the plan and give feedback, or is it to refine the plan plan with local knowledge into a more detailed plan? The modalities required to answer these questions are usually defined as *performatives* that describe communicative acts and form part of an agent communication message. Statements describing factual knowledge or plans (built using the ontology and formal language) are usually used at the content level, i.e. they provide the object of a message using a given performative.

We will now look at the performatives defined in the two efforts mentioned above, KSE and FIPA. More specifically, we will look at the performatives they define that can be used to support plan execution and agent communication about plans.

15.1 Knowledge Sharing and KQML

In the Knowledge Sharing Effort, the language for expressing ontologies is called Ontolingua, and the formal language for representing content is the Knowledge Interchange Format (KIF). The language for expressing complete agent communication messages based on different performatives and containing content in KIF (or some other content language) is the Knowledge Query and Manipulation Language (KQML) [Labrou and Finin, 1997]. Before we look at the performatives defined in KQML, we shall have a brief look at the structure of a KQML message. This will clarify the relation between a KQML message and its content, which is not limited to KIF, but could also be expressed in a different language such as a CPR/BDI-based plan representation.

In the KSE framework, agents can send messages to each other and KQML is a standard that defines the syntax of a single message. As the language is based on a LISP-like syntax, a message is a list of symbols surrounded by brackets:

```
<message> ::= ( <performative>
                { <keyword> <value>}* )
```

The first element in this list is the performative that defines what type of communicative act this message represents. The performative describes what the sender wants the receiver to do with the content. If the content is factual, for example, the sender may use the `tell` performative to indicate that the receiver should (from now on) believe the given fact. If the performative was `ask`, the sender will expect a reply indicating whether the receiver believes the given fact. If the content is a plan a different set of performatives may be used, e.g. to tell the receiver to execute, evaluate, or refine the given plan.

The remainder of a KQML message are alternating keywords and values. KQML defines a number of keywords that can be used in a message:

- `:sender`: the sender of this message;
- `:receiver`: the receiver of this message;
- `:from`: the original sender of a forwarded message;
- `:to`: the final receiver of a forwarded message;
- `:in-reply-to`: a label that allows to tie this message to a previous message;
- `:reply-with`: a label that allows to tie this message to a future message;
- `:language`: the name of the representation language used in the content;
- `:ontology`: the name of the ontology used in the content;
- `:content`: the content with respect to which this performative is applied.

The first four keywords are used to identify the agents participating in the communication. The next two can be used to put this message into a larger context: a conversation consisting of multiple messages.

For the present discussion the last three keywords are the most relevant. What KQML provides here is a mechanism that allows for the plugging in of a different language for the content, that is, for the direct object of the communicative act. For example, if the message is about a plan, this plan can be sent as the content of a message. The ontology explicitly references the set of terms that will be used in the content, which defines a meaningful vocabulary. In a planning context this could be the CPR/BDI ontology. The language explicitly specifies the formal language used to represent the content. This could be an XML-based language or a LISP-like syntax[2], for example. If the message was about factual information, the content language would presumably have to be a different language, e.g. KIF.

To summarize, a KQML message consists of a performative defining the modality of the message and a set of keyword-value pairs defining to agents involved, references to the context, and the content of the message which can be given in an explicitly specified representation language that is not defined as part of KQML.

15.1.1 Performatives

Given our focus on distributed planning and execution, we are most concerned with agent communication where the content of a message is a plan, or at least activity-related. We shall now have a look at the performatives defined in KQML to see which communicative acts related to distributed planning and execution are available in this language. The complete list of performatives defined in KQML is given in Appendix A. These performatives can be divided into three groups, depending on the type of objects (the content) the communicative acts are about:

- Messages about facts: Performatives from this group allow agents to interact with each other's knowledge, either telling them about facts that they believe to hold, or asking them about facts. If the answer consists of many parts, it can be streamed rather than sent in one large message.

[2]While the syntax of the content language is not constrained in KQML, the problem of parsing a complete message means that it must be possible to at least recognize the end of the content somehow.

- Messages about activities: Performatives from this group allow agents to ask each other to achieve given goals. The achievement clearly involves the execution of activity, although this is not explicitly mentioned in the message. Furthermore, since the content of the message is a goal, the representation of plans is not required.

- Messages about capabilities: Performatives from this group allow agents firstly to register capabilities with a central capability broker, and secondly to find agents that have required capabilities. The broker may then manage the application of a capability.

The first set of performatives is not activity-related at all and can be ignored for the present discussion.

The second set indirectly deals with activities. With the `achieve` performative the sender asks the receiver "to want to try to make the content true of the system". We shall interpret this as the setting of a goal in the classical planning sense, i.e. the receiver should come up with and execute a plan that will achieve said goal in the world state. Presumably this will require the receiver to perform some actions. However, the actions themselves are not subject to agent communication in KQML. The other performative that relates to activity is `unachieve`, which should only be used after an `achieve` relating to the same goal. From a planning perspective this could be the same as another `achieve` message with a negative goal. `achieve` and `unachieve` are the only performatives dealing with goals.

The third set can be used for capability brokering. The central performative here is `advertise` with which an agent can announce that it is capable of processing a given type of message. While capabilities in general deal with activities, a capability advertisement in KQML only describes the message type that can be processed, i.e. the content will itself be another KQML message and not an activity or a plan. Thus, the performatives from this set cannot be considered to be activity-related in KQML.

15.1.2 Protocols

Messages between agents are intended to appear as part of a dialogue or a larger communication structure. A protocol is such a structure and it can be described as a communication plan schema, i.e. all the actions in the plan schema are communication actions using the KQML performatives. Protocols are schemata in the sense that an actual communication represents an instantiation of the schema.

While the KQML specification describes the context in which certain message types may occur, there is no formal specification that describes the interactions that may take place.

15.2 The FIPA Agent Communication Language

The Foundation for Intelligent Physical Agents (FIPA) has proposed an alternative standard for an Agent Communication Language (ACL). As for KQML, we shall first look at the message structure defined in FIPA ACL and then the set of performatives defined in this standard, focussing on those that deal with activity management.

15.2.1 Message Structure

The structure of a FIPA ACL message[3] is very similar to that proposed in KQML. Each message must have a performative that describes the type of communicative act the message represents. The remainder of the message contains fields for describing the participants in the communication, tying the message into a conversation consisting of multiple messages, and the actual content of the message. In detail, a FIPA ACL message contains:

- `performative`: denotes the type of the communicative act of the ACL message;

- `sender`: denotes the identity of the sender of the message, that is, the name of the agent of the communicative act;

- `receiver`: denotes the identity of the intended recipients of the message;

[3]See http://www.fipa.org/repository/standardspecs.html for the set of documents describing the FIPA standard.

- `reply-to`: indicates that subsequent messages in this conversation thread are to be directed to the agent named in the reply-to parameter, instead of to the agent named in the sender parameter;
- `content`: denotes the content of the message; equivalently denotes the object of the action: the meaning of the content of any ACL message is intended to be interpreted by the receiver of the message;
- `language`: denotes the language in which the content parameter is expressed;
- `encoding`: denotes the specific encoding of the content language expression;
- `ontology`: denotes the ontology(s) used to give a meaning to the symbols in the content expression;
- `protocol`: denotes the interaction protocol that the sending agent is employing with this ACL message;
- `conversation-id`: introduces an expression (a conversation identifier) which is used to identify the ongoing sequence of communicative acts that together form a conversation;
- `reply-with`: introduces an expression that will be used by the responding agent to identify this message;
- `in-reply-to`: denotes an expression that references an earlier action to which this message is a reply;
- `reply-by`: denotes a time and/or date expression which indicates the latest time by which the sending agent would like to receive a reply.

Like KQML, the fields for describing the participants include the sender and receiver. The fields to do with message forwarding are not supported in FIPA. Instead, the `reply-to` field can be used to specify a different response address. Conceptually, these differences are hardly significant.

Similarly, the fields for conversation management are not fundamentally different: two additional fields are defined in FIPA, `conversation-id` and `reply-by`, where the latter allows the specification of a deadline at the envelope level, something that has to be done as part of the message content in KQML.

Finally, the language used for the content is not defined in the standard and various fields in a message specify what formal structure is used. Again, the ontology and the formal language can be named explicitly. In addition, FIPA ACL allows the explicit naming of the protocol that underlies the conversion to which a message belongs.

15.2.2 Performatives

The complete list of performatives defined in FIPA ACL is given in Appendix B. These can be roughly divided into three sets based on the type of content and the protocols these performatives are expected to be used in:

- Messages about facts: Performatives from this group are concerned with knowledge manipulation where each agent has its own knowledge-base and agent communication is used to update and query information across agent knowledge-bases.
- Message about collaboration: Performatives from this group can be used to organize a collaboration between agents. Essentially, these performatives implement the Contract Net protocol [Smith, 1980].
- Messages about communication management: Performatives from this group allow message forwarding and a single performative that can be used to indicate that a received message was not understood.

As for KQML, the first set deals with factual knowledge and not with activity. The content of messages using these performatives could again be expressed in KIF or some other logical language.

The second set corresponds to the brokering-related performatives in KQML. However, whereas KQML used itself as a content language (defining the type of message that can be processed), FIPA ACL is based on the Contract Net protocol in which the central element is a proposal for work. Thus, the content of most messages must be about such a proposal. Unfortunately neither the document that specifies the performatives nor the document that defines the contract net interaction protocol defines what a proposal should look like or what type of representation might be used here[4]. Two options, at least from an AI planning perspective, would be to use a goal-based or a task-based representation, but this is only speculation. Another issue with the performatives provided by FIPA is that they focus on the phase leading up to a collaboration, but provide little for managing the distributed activities that implement the collaboration. No performatives related to plan sharing and execution are defined in FIPA ACL.

The third set of performatives are not related to activities and thus not of interest to us.

15.2.3 Protocols

An area in which the FIPA standard is clearly more advanced than the KSE specifications is the set of interaction protocols that define how single messages can be used in conversations. FIPA defines the following interaction protocols in detail:

- FIPA Request Interaction Protocol Specification: allows one agent to request another to perform some action.
- FIPA Query Interaction Protocol Specification: allows one agent to request to perform some kind of action on another agent.
- FIPA Request When Interaction Protocol Specification: allows an agent to request that the receiver perform some action at the time a given precondition becomes true.
- FIPA Contract Net Interaction Protocol Specification: a minor modification of the original contract net IP pattern in that it adds rejection and confirmation communicative acts.
- FIPA Iterated Contract Net Interaction Protocol Specification: an extension of the basic FIPA Contract Net IP, but it differs by allowing multi-round iterative bidding.
- FIPA Brokering Interaction Protocol Specification: designed to support capability brokerage interactions in mediated systems and in multi-agent systems.
- FIPA Recruiting Interaction Protocol Specification: designed to support recruiting interactions in mediated systems and in multi-agent systems.
- FIPA Subscribe Interaction Protocol Specification: allows an agent to request a receiving agent to perform an action on subscription and subsequently when the referenced object changes.
- FIPA Propose Interaction Protocol Specification: allows an agent to propose to receiving agents that the initiator will do the actions described in the propose communicative act when the receiving agent accepts the proposal.

A number of these protocols deal with capability brokering and are thus relevant to distributed planning and plan execution. However, only one of the protocols deals directly with the distributed execution of activity, namely the FIPA Request interaction protocol. An overview of the messages to be exchanged as part of this protocol is given in figure 12.

The interaction protocol involves two agents, the *initiator* that wants another agent to execute some action, and the so-called *participant* that executes the action. The first step in the interaction protocol consists of the initiator sending a request message to the participant, that is, a message with the `request` performative and an action as the content of the message. What language the action is to be described in is not specified in the protocol and the idea is, of course, that the outer layer can encapsulate different content languages.

The participant has to respond to this message with either a `refuse` or an `accept` message. In the former case the protocol is terminated and no further messages will be exchanged. In the latter case the interaction continues. Presumably, the participant will execute the requested action and then send a message indicating the status of the action to the initiator. Depending on the outcome of the action, the message can be a `failure` message or an `inform` message. This message terminates the interaction.

This basic protocol only shows the flow of messages when things go according to plan, so to speak. Either participant may also terminate the protocol when they receive a message they cannot pro-

[4]FIPA also includes a specification for a content language, the FIPA SL content language, but it is not obvious how a protocol might be expressed in this language.

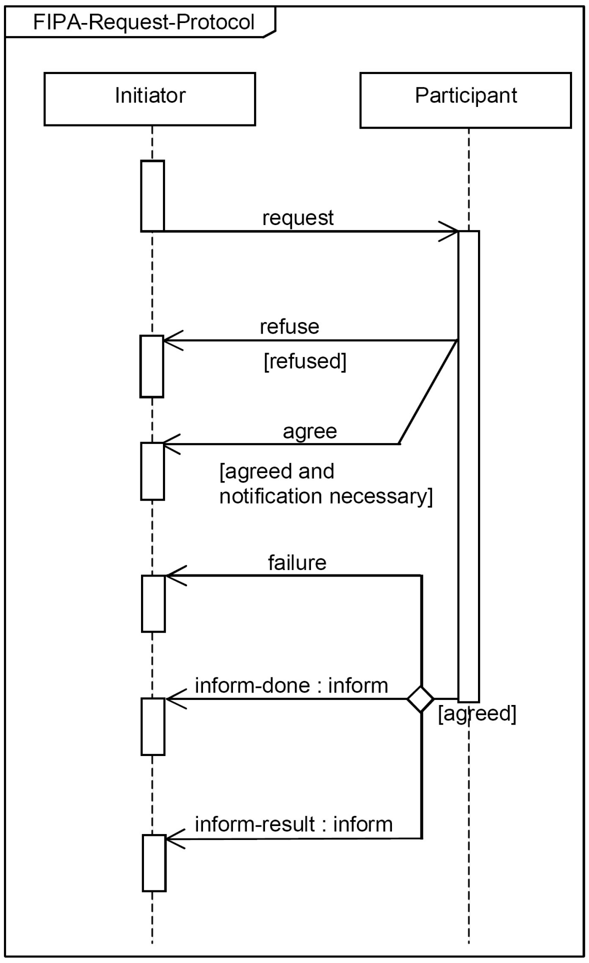


Figure 12: FIPA Request Interaction Protocol

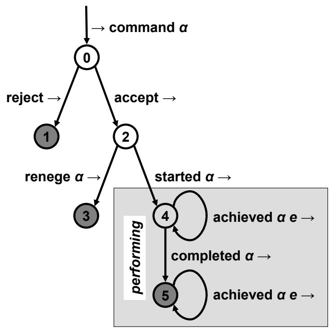


Figure 13: Protocol for Activity Execution

cess, by sending a `not-understood` message. Furthermore, the initiator can send a `cancel` message, asking the participant to abandon execution of the action. How this is possible depends on the action, of course.

16 Agent Communication for Plan Execution

In this section we will extend the basic `request` interaction protocol defined in FIPA to define more complex protocols that allow for more detailed control of the execution, and which build on some of the features of the CPR/BDI representation that may be used at the content level.

16.1 Simple Action Execution

The first protocol we have developed is a slightly more complex version of the FIPA request interaction protocol that differs from the FIPA protocol in two ways: firstly, it allows the executing agent to renege after it has accepted, and secondly, the protocol is more detailed with respect to the information that indicates the status of execution to the requester. A concise overview of the protocol is shown in figure 13.

This protocol is shown here from the perspective of the action-executing agent. A corresponding version for the requester would look very similar. Nodes in this graph represent internal states of the agent with respect to the protocol. Arcs represent possible state transitions that can occur due to a message being sent or received. The message is given as a label on the arc. A message being received is indicated by an arrow before the message and a message being sent is indicated by an arrow after the message. States in which the protocol can be terminated are shown gray.

The protocol is initiated with an incoming message in which a command is issued. This essentially corresponds to the request message in FIPA. The terminology is different to indicate that this message represents the issuing of a command rather than a request. This indicates that the sender must have authority to issue commands to the receiver, which may be due to a pre-existing authority relationship or some prior negotiation that has taken place (by means of another protocol not defined here).

Given that the sending agent has authority over the receiving agent, the expected response would be for the receiver to accept. However, there may be a reason why the receiver has to reject, for example, if a resource required for the action execution is currently unavailable. Thus, the executing agent has to respond to accept or reject the command.

It is not assumed that commands are only issued for immediate action. Normally, there would be a time interval specified as part of the activity description: CPR/BDI allows for at least two time points to be specified with any activity, beginning and end of the activity. This means that things can happen after the acceptance of a command and the start of the corresponding action. If something happens that means the receiver can no longer execute the command, it has to send a renege message to the agent that issued the command.

When the executing agent begins with the execution, it has to send an according message to the agent that issued the command. At this point the executing agent is said to be "performing", a block that will be reused in later protocols and thus indicated as a box in figure 13.

In CPR/BDI, as in most AI planning representations, an action is associated with effects, that is, fluent relations in the world state that change as a direct result of the action being executed. Given that we consider actions to be temporal, effects can occur at different time points, but not before an action has been started. Some effects will occur immediately after the beginning of an action (especially negative effects, e.g. when a robot moves away from a location) and others may occur later. Some actions even have effects after the end of the execution (e.g. the paint being dry after the execution of a paint action).

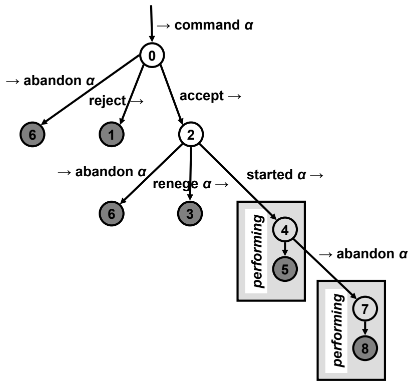


Figure 14: Abandoning Execution

In the protocol described here the executing agent is expected to send progress reports to the commanding agent every time and effect is achieved and when the action has been completed. Assuming a shared model of the action the commanding agent can then monitor the progress and knows when the effects can be used by other agents and actions, or when execution deviates from the model, which could be considered a failure. Note that this is a far more detailed model of a failure than the simple message in the FIPA protocol.

16.2 Abandoning Execution

In an extension to the basic protocol the FIPA specification allows the commanding agent to send a cancel message. The executing agent is meant to somehow stop executing the command at this point. Figure 14 shows an extended version of the protocol described above to allow for similar functionality: abandoning the action at any stage.

The protocol is initiated as before by the commander sending a command to the executing agent. Assuming that the executing agent does not respond immediately (presumably it will have to do some planning before it can accept), it is possible that the commander cancels the command before the executing agent accepts or rejects. This would terminate the protocol and should not lead to further complications.

If the executing agent rejects the protocol is terminated, as before. If it accepts, the commander may still cancel the command before the execution begins. Again, this should not lead to further complications and simply terminate the protocol.

However, things become more complex once the execution has started, and the commander will be aware of this because it has received an according message from the executing agent. If the commander does not issue a message asking for the abandonment of execution, the protocol proceeds as before: the executing agent sends messages when effects have been achieved and when the action has been completed. Once the action has been completed it can no longer be abandoned, even if there are delayed effects still taking place.

How an action can be abandoned during execution clearly depends on the action in question. We shall assume here that there are various points during an execution at which an action can be abandoned. Clearly, effects that have already occurred at the point of abandonment are not automatically undone. However, future effects may or may not be achieved. In fact, there may be a different set of effects that has to be expected depending on the point at which an action is abandoned. The protocol we have defined is based on this model: normal performing is abandoned, but the performance of some other action now continues. This will result in effects being achieved and these will be relayed through messages to the commander, including the completion of the changed activity.

Ideally, the commander and the executing agent have a shared model of activity that specifies how actions can be abandoned. This could be done through one set of effects for normal execution, and various sets of additional effects associated with different time points at which the action may be abandoned. If the action cannot be abandoned, the model collapses into the classical model: an action with a set of preconditions and effects. Given such a shared model it would be possible for the commander to anticipate the result of abandoning an action and hence, making a more informed decision.

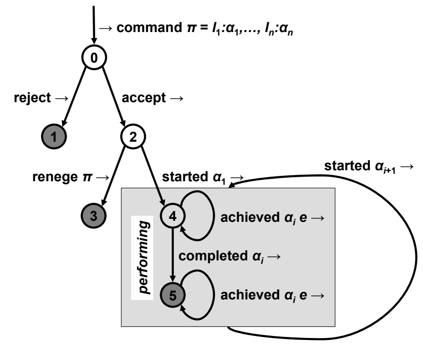


Figure 15: Protocol for Plan Execution

16.3 Execution of Plans

The next step to extending the protocol involves the commanding agent sending a whole plan to the executing agent. For simplicity, this plan is assumed to be a sequence of actions. Each action in the plan must have a unique label so that it can be uniquely identified in the plan. The protocol outline is shown in figure 15, again from the perspective of the executing agent.

As before, the protocol begins with the commander issuing the command, which is a plan. The executing agent can either reject or accept this plan, where acceptance is the expected behavior. Also, the executing agent can renege before the start of the first action in the plan. Extending the protocol with cancelation messages should be straight forward.

Once the execution of the first action in the plan begins, a message has to be sent informing it that the action has been started. This followed by messages about achieved effects and completion of the first action. This is effectively the performance block introduced in section 16.1. In fact, there will be one such block of messages corresponding to every action in the plan. This allows the commanding agent to follow execution of the plan in detail.

One obvious question here is how this differs from a sequence of single execution commands issued by the commander. The main difference is that the executing agent sees the whole plan at once. Thus, if a resource for an activity later in the plan is not available, the executing agent can reject the plan

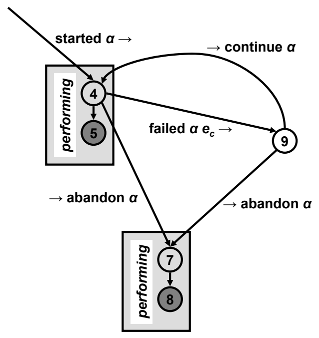


Figure 16: Protocol for Execution Failure

before the execution of the first action. On the other hand, if the commander wants to change the plan during execution, and extension of the protocol would be required allows for appropriate intervention. In the simplest case this might be like abandoning execution of the current action, and therefore quite similar to the protocol described in the previous section.

17 Dealing with Execution Failure

The next obvious question then is how to deal with execution failure. By execution failure we mean any situation in which the executing agent fails to achieve the effects that are in the shared model of action.

17.1 Execution Failure Protocol

A partial protocol for dealing such failures is shown in figure 16. The part up the execution of the action is not included as this cannot lead to an execution failure.

The protocol is shown from the message sent to the commander that indicates that execution has begun. This may lead to normal execution and messages being exchanged as described in the "performing" block defined above. Let us assume that, at some point after the start of the action, execution fails. This means an expected effect did not occur. This can mean that either nothing has been achieved or, that an alternative outcome occurred. In either case, the executing agent sends a message to the commander informing it that it failed to achieve an expected effect.

The commander can then decide whether it wants the executing agent to continue with the execution or abandon the action. This assumes that the commander is aware of the goal structure and can decide whether the effect in question was required for a subsequent action. If it was not, the failed effect will not influence the plan. If the effect was required by another action the commander will need to do more than just tell the executing agent to abandon execution.

17.2 Execution Failure and Planning

Execution failure is not a new problem in AI planning, with plan repair and re-planning being the usual approaches. However, these approaches are of limited applicability as they are only dealing with expectable failure.

The underlying problem is really one of abstraction. When setting up a classic planning problem in the first place, it is important to abstract away from unnecessary detail in order to keep the search space as small as possible. Classic AI research illustrates how this can be done [Amarel, 1968]. As a result, each state in the search space corresponds to many states in the world state.

This may be a problem when execution fails: the world state in which an agent finds itself may correspond to a search state that represents successful execution of an action, or it may not have a corresponding abstract state in the search space. For example, in a route planning problem we may only consider specific locations in states, e.g. cities. Then, no matter where in a city the agent is, the search state representing this world state is the same. And if an agent fails to get from one city to another, its actual position will be somewhere along the way, which does not correspond to a search state at all. Thus, a planner would never find a plan that recovers this agent, unless the underlying planning problem that defines the search state space is changed.

So, on the one hand we need to abstract away from detail to create a small search space, but then when plan execution fails, we may need to be able to include more detail to deal with the failure at hand. Solving this problem is beyond this report.

18 Results and Discussion

This report describes work completed towards a new framework for distributed multi-agent planning and plan execution. The first step has focussed on the plan representation that needs to be sufficiently rich to allow the sharing of plans between humans, software systems, and robotic entities. The basis for this representation is the Core Plan Representation discussed in section 3 and the BDI model of agency discussed in section 4.

The main result of this work then is a combined and refined ontological model for rich plan representations. This model includes concepts that will be required for a sharable plan representation intended for distributed multi-agent planning and plan execution.

While the first part of this report does specify the ontology that constitutes the merged representation, it does not specify a specific syntax for the representation. Furthermore, a meta-level will be required for communicating plans, allowing agents to not only share plans, but also to tell other agent what they expect them to do with the communicated plan. This will require a set of performatives for working with shared plans.

This next part of this report addresses this shortfall. It describes an implementation of the CPR/BDI framework that comes in the form of an extension to the MediaWiki software. This extension defines a number of XML tags that can be used to mark up procedural knowledge in a wiki article, resulting in a semi-formal representation overall. The advantage of using MediaWiki is that it implements Web 2.0 style information sharing in a robust framework. The extension we provide makes it possible to represent procedural knowledge with enough formalism to make it accessible to reasoning engines such as AI planners.

This report has also defined four planning domain features that can be used by knowledge engineers to provide information about the domain they are encoding. The formal definition of the features was used to design algorithms that can extract the actual feature values from the domain description. The algorithms are based on the domain description only, i.e. they do not require a planning problem as input. The extracted features can then be compared to the feature values specified by the domain author to validate the domain description. This approach has been evaluated using domains taken mostly from the international planning competition. The result shows that features were consistent with those available in the domains, where explicitly specified. Those features that were not specified were extracted and manually verified, to ensure they are consistent with the given set of operators.

The first feature, the type system, is a rather simple, flat division into equivalence classes. This may not be suitable for very complex planning domains, but the domains we have analyzed do not exhibit much hierarchical structure. The advantage of such a type system is that it can be easily added to the operator descriptions in the form of unary preconditions. Furthermore, we showed that the type system derived by our algorithm is the most specific type system of its kind based solely on the operator descriptions. An open question is whether this is identical to the least general generalization [Plotkin, 1969] used in machine learning. The algorithm could be refined to derive a hierarchical type system if one takes into account the directionality of the operators, but for a type system consisting of equivalence classes this is irrelevant. Also, the algorithm described in this paper should also be applicable to hierarchical task network domains, but this has not yet been implemented.

Actions with inconsistent effects are another feature we have defined. For most domains, such actions are probably not desirable. In fact, the admission of such actions leads to a different planning problem as the state spaces with or without such actions may be different for the same planning domain and problem. Also, planners that translate a STRIPS planning problem (with negative preconditions) into a propositional problem (without negative preconditions) need to be more careful if actions with inconsistent effects are permitted. The translation method described in [Ghallab *et al.*, 2004, section 2.6] does not work in this case as it introduces independent predicates for a predicate and its negations, which can become true in the same state if an action with inconsistent effects is applied. This would render the planner potentially unsound.

The final feature which defines reversible actions is somewhat different as it can only be usefully used as a necessary criterion to test whether one operator is the reverse of another. The more strict, sufficient definition does not provide any computational advantage. The difference is simply that the necessary criterion can be computed on the basis of the operator descriptions, whereas the sufficient test requires knowledge of the state in which an action is applied. The difference is quite subtle though, and may not matter in practice. The necessary criterion requires the positive and negative effects to cancel each other. However, if a state contains an atom that is also added by the first action, but then deleted by the second action, then the state will be changed. If an operator listed all the relevant atoms also as preconditions, this exception would not hold.

Finally, we have looked at established agent communication frameworks with the focus on support they might provide for distributed plan execution. The message structure in the analyzed frameworks is rather similar and seems sufficiently expressive. Much rests on the performatives in the two frameworks as these define the communicative acts to the agents. However, communicative acts do occur in isolation, and it is the interaction protocols that defines the context in which performatives are to be used. The FIPA agent framework is more advanced in this respect and comes closer to our needs by defining a protocol specifically for the distributed execution of an action. This protocol has formed the basis for our own protocols described in this report.

Having defined a number of protocols it is now possible to go back and analyze them in terms of communicative acts required. The following is a list of all the message types used in the protocols for distributed plan execution:

- `command`: the commanding agent issues a command; the content is a description of the commanded activity; corresponds to the `request` performative in FIPA;
- `reject`: the executing agent rejects the command; no content required; corresponds to the `refuse` performative in FIPA;
- `accept`: the executing agent accepts the command; no content required; corresponds to the `agree` performative in FIPA;
- `renege`: the executing agent reneges the command; the content is a reference to the action in question; no corresponding performative in FIPA;
- `started`: the executing agent starts the execution; the content is a reference to the action in question; could use `inform` performative in FIPA to convey this message;
- `achieved`: the executing agent has achieved and effect; the content is a reference to the effect of an action; could use `inform` performative in FIPA to convey this message;
- `completed`: the executing agent has completed the execution; the content is a reference to the action in question; could use `inform` performative in FIPA to convey this message;
- `abandon`: the commander instructs the execution agent to terminate execution (now); the content is a reference to the action in question; corresponds to the `cancel` performative in FIPA;
- `failed`: the executing agent has failed to achieve an effect; the content is a reference to the effect of an action;could use `inform` performative in FIPA to convey this message;
- `continue`:the commander instructs the executing agent to continue with the execution; the content is a reference to the action in question; no corresponding performative in FIPA;

As can be seen, the majority of the performatives required is already used for the simplest protocol. This indicates that the set of performatives required for even more complex protocols could be quite limited, rendering an implementation feasible.

19 Conclusions

Generative planning is a computationally difficult problem. Putting plans in a wider context is a conceptually difficult problem in the sense that there is no single, agreed-upon definition of the plan communication or execution problem. The report describes work towards a framework that addresses these problems, even if there is no formal definition.

The first step has been an ontological analysis of a plan. If plans are to be communicated, it needs to be agreed what constitutes a plan, and an ontological view defining relevant concepts is the normal way of going about this. We have implemented this view in the context of a wiki, allowing for shared and distributed knowledge engineering of planning knowledge. One of the most interesting lessons learned from this work is perhaps the fact that there is much in the formal definition of a planning domain, that is not explicit. This is a serious problem when it comes to knowledge sharing, or maintaining the formal representation. To address this problem we have developed a number of algorithms that analyze a domain in terms of features that could also be specified in the formal definition. If these formal definition and the extracted features are different, this indicates a problem with the domain definition. Also, the explicit representation will help others to better understand the domain.

The ontology developed in the first part is for describing plans. The meta-level developed in the final part of the project defines a number of performatives and protocols that can be used during the execution of a plan. This should make it possible to control the execution at least to a limited degree in a formal way.

References

[Allen *et al.*, 1990] James Allen, James Hendler, and Austin Tate, editors. *Readings in Planning.* Morgan Kaufman, 1990.

[Amarel, 1968] Saul Amarel. On representations of problems of reasoning about actions. In Donald Michie, editor, *Machine Intelligence 3*, pages 131–171. Elsevier/North-Holland, 1968.

[Barrett, 2008] Daniel J. Barrett. *MediaWiki: Wikipedia and Beyond.* O'Reilly, 2008.

[Blum and Furst, 1995] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proc. 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1636–1642. Morgan Kaufmann, 1995.

[Coles and Smith, 2006] Andrew Coles and Amanda Smith. Generic types and their use in improving the quality of search heuristics. In *Proc. 25th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2006)*, 2006.

[Conklin, 2005] Jeffrey Conklin. *Dialogue Mapping: Building Shared Understanding of Wicked Problems.* John Wiley and Sons, 2005.

[Dawson and Siklossy, 1977] Clive Dawson and Laurent Siklossy. The role of preprocessing in problem-solving systems. In *Proc. 5th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 465–471. Morgan Kaufmann, 1977.

[Fikes and Nilsson, 1971] R. Fikes and N. Nilsson. STRIPS: A new appraoch to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971. Reprinted in [Allen *et al.*, 1990, pages 88–97].

[Fox and Long, 1998] Maria Fox and Derek Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.

[Fox and Long, 2003] Maria Fox and Derek Long. PDDL2.1 : An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[Ghallab *et al.*, 2004] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning.* Morgan Kaufmann, 2004.

[Kambhampati *et al.*, 1995] S. Kambhampati, C. Knoblock, and Q. Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence*, 76:167–238, 1995.

[Labrou and Finin, 1997] Yannis Labrou and Tim Finin. A proposal for a new KQML specification. Technical Report TR CS-97-03, University of Maryland Baltimore County (UMBC), Baltimore, Maryland, February 1997.

[McCluskey and Porteous, 1997] T.L. McCluskey and J.M. Porteous. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence*, 95:1–65, 1997.

[Pease and Carrico, 1996] R. Adam Pease and Todd M. Carrico. Object model working group core plan representation. Technical Report AL/HR-TP-1996-0031, United States Air Force Armstrong Laboratory, Wright-Patterson AFB, OH, 1996.

[Plotkin, 1969] Gordon Plotkin. A note on inductive generalization. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 5*, pages 153–164. Edinburgh University Press, 1969.

[Rao and Georgeff, 1991] Anand Rao and Michael Georgeff. Modeling rational agents within a BDI-architecture. In *Proc. 2nd International Conference on Knowledge Representation and Reasoning (KR)*, pages 473–484. Morgan Kaufmann, 1991.

[Russell and Norvig, 2003] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2nd edition, 2003.

[Smith, 1980] R. G. Smith. The contract net protocol: High level communication and control in a distributed problem solver. *In IEEE Transactions on Computers*, C-29(12):1104–1113, 1980.

[Tate, 1977] Austin Tate. Generating project networks. In *Proc. 5th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 888–893. Morgan Kaufmann, 1977. Reprinted in [Allen *et al.*, 1990, pages 291–296].

[Tate, 2003] Austin Tate. <I-N-C-A>: A shared model for mixed-initiative synthesis tasks. In Gheorghe Tecuci, editor, *Proc. IJCAI Workshop on Mixed-Initiative Intelligent Systems*, pages 125–130, 2003.

[Wickler *et al.*, 2006] Gerhard Wickler, Stephen Potter, and Austin Tate. Recording rationale in <I-N-C-A> for plan analysis. In Lee McCluskey, Karen Myers, and Biplav Srivastava, editors, *Proc. ICAPS Workshop on Plan Analysis and Management*, pages 5–11, 2006.

[Wickler *et al.*, 2007] Gerhard Wickler, Stephen Potter, Austin Tate, Michal Pěchouček, and Eduard Semsch. Planning and choosing: Augmenting HTN-based agents with mental attitudes. In *Proc. International Conference on Intelligent Agent Technology*, 2007.

[Wickler, 2010] Gerhard Wickler. Plan representations for distributed planning and execution. Technical Report Project Report 1, AIAI, University of Edinburgh, Edinburgh, Scotland, 2010.

[Wooldridge and Jennings, 1995] Michael Wooldridge and Nicholas Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[Wooldridge, 1999] Michael Wooldridge. Intelligent agents. In Gerhard Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, chapter 1, pages 27–77. The MIT Press, 1999.

List of Symbols, Abbreviations, and Acronyms

AI - Artificial Intelligence
BDI - Beliefs, Desires and Intentions
CPR - Core Plan Representation
HTN - Hierarchical Task Network
PDDL - Planning Domain Definition Language